Aurelia IN ACTION

Sean Hunter

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Aurelia in Action
Version 2

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Aurelia in Action*. I first came across Aurelia in a 2015 .NET Rocks podcast interview with Rob Eisenberg (the creator of the framework). Rob discussed a new, next-generation single-page application (SPA) development framework that took advantage of the latest features in ECMAScript (ES) and web-component specifications. My interest was piqued. As I started digging into the framework I quickly noticed the differences between Aurelia and the previous SPA frameworks I'd worked with. Two things struck me as I began to explore the framework: first, it's clean. The verbose syntax and noise that I expected to see simply weren't there. Second, it was written using new web technologies such as ES6 and 7, which I'd not previously had much experience with. I was hooked. It wasn't long before I was using Aurelia in production, and became known within the company where I was working as somewhat of a framework evangelist. In this book, I want to share with you all the things that excited me when I first started working with Aurelia. My goal isto give you the knowledge to build your own exciting projects. If you can exasperate your own teammates with your enthusiasm, all the better!

Learning Aurelia is about learning modern web development. I'm not assuming any prior knowledge of ES6 or ES7 (often called ESNext) or modern browser features such as HTTP Fetch or web components; you'll learn them as you go. If you can build a web application with standard server-side technologies such as ASP.NET MVC, Ruby on Rails, Node.js, or JSP and can validate forms with JavaScript—or if you've used an SPA framework such as AngularJS or Backbone.js—you've got all the pre-requisite knowledge you need to breeze through the concepts in this book.When you're finished, you'll have working knowledge of modern web-development technologies such as ESNext and web components, which will be valuable in your web development journey.

Throughout this book, I'll take you on a tour of the Aurelia framework. In Part 1, you learn about the Aurelia architecture and philosophy and how it's different from other SPAs, and then you build a basic application. In Part 2, you take a deep dive into the main primitives that make up the framework and review how to use them in the context of a non-trivial SPA. Finally, in Part 3, you learn what it takes to run Aurelia in a real-world environment, and how to test, bundle, and package an application for deployment.

In this update, I've refined chapter 1 and 2, tightening the framework overview to improve the overall reading flow. I've also added chapters 3, 4 and 5, which take you through the main components of Aurelia's templating and binding system, from custom elements and attributes, to value converters and binding behaviors.

Before you begin, I want to ask favor: please provide any feedback that you think of on the *Aurelia in Action* online forum. As an early access member, you've shown your excitement and interest in the topic. The feedback provided so far has been invaluable, and many of the

comments submitted on the forum have been incorporated back into the book already, helping to make this book the best learning experience it can be.

—Sean Hunter

brief contents

PART 1: INTRODUCTION TO AURELIA

1 *Introducing Aurelia*

2 *Building your first Aurelia application*

PART 2: EXPLORING AURELIA

3 *View resources, custom elemements, and custom attributes*

4 *Aurelia templating and binding*

5 *Value converters and binding behaviors*

6 *Inter-component communication*

7 *Working with forms*

8 *Working with HTTP*

9 *Routing*

10 *Aurelia view composition*

11 *Animating Aurelia*

12 *Extending Aurelia*

13 *Aurelia and web components*

PART 3: AURELIA IN THE REAL WORLD

14 *Deploying Aurelia applications*

15 *Testing*

16 *Putting it all together*

APPENDIX

A *Installation & set up*

1 Introducing Aurelia

This chapter covers

- What Aurelia is, what it's not, and why you should care
- The kinds of applications suited to development using the Aurelia framework
- An overview of what you'll learn in this book
- A tour of the Aurelia framework

Aurelia is a front-end JavaScript framework focused on building rich web applications. Like other frameworks such as Angular, and EmberJS, Aurelia is a single-page application development framework (SPA). This means that Aurelia applications deliver the entire user experience on one page without requiring the page to be reloaded during use. At its core writing an Aurelia application is just writing a JavaScript application. But, Aurelia applications are written with the latest versions of JavaScript (ES2015 and beyond, which we'll dig into as we go along, or Typescript). The Aurelia framework has all the tools that you need to build the rich and responsive web applications that users expect today, using coding conventions closely aligned to web standards.

1.1 Why do I care?

Imagine you're having a Facebook chat session with your friend Bob, and every time you send a message you need to pause for a 3-second delay while the page reloads to show you whether you've successfully sent the message, whether Bob has received it, and whether you've received any other messages from Bob in the meanwhile. In this scenario, it would be difficult to have a fluid conversation because of the jarring pause between entering your message and receiving feedback from the application. You may ask Bob a question, only to find that by the time that the page reloads he's already answered it. In reality, the experience

is much different. As soon as you start typing a message Bob can see that you're composing a message for him, and when you click *send* you receive visual feedback in the form of tick to indicate that the message was delivered successfully. It's easy to gloss over functionality like this today because it's a part of so many of the modern applications we use all the time such as Slack, Skype, or Facebook messenger.

Now imagine that you've been tasked with building a line of business application for your department. The HR department has implemented an employee-of-the-month system where staff nominate and vote on who most deserves a monthly prize. This app would be expected to provide things like the following:

- A responsive voting system
- Live updating charts showing an overview of who has the most votes
- Validation to prevent employees from voting more than once

> **DEFINITION** Responsive web applications can be used across a variety of devices, from smart phones to desktop PCs. Typically, this is achieved using CSS media queries to re-size various sections of the page or even hide them entirely so that the user experience is optimal for the device at hand.

The features listed for our HR application are common examples of the kinds of things that users – like our fictional HR department – expect in rich web applications. Rich web applications such as Facebook and Slack have raised the bar in terms of what users expect from *all* web applications they use. A trend I've noticed over the past few years is that clients have begun to expect the same kind of richness out of a line-of-business application that they are used to seeing in applications they use outside of the office. Using an SPA framework like Aurelia makes it vastly simpler to build these kinds of applications, compared with the traditional request/response style of architecture used with frameworks such as ASP.NET MVC, JSP, or Ruby on Rails, to name a few.

Given that we want to create rich, responsive web applications, the next logical question is, which technology should we use to do this? A useful technique for answering this kind of question is to analyze the kinds of attributes that are important to you and your team with a set of questions like this:

- **How complicated are the applications we are trying to build?** We don't want to use a jack-hammer to crush a walnut when a nut-cracker will do the job just fine. Conversely, we want to make sure that we have a sufficient foundation in place that will support the kind of application we are trying to build.
- **How important is web standards compliance to the team?** A framework that adheres more closely to web standards is more likely to look familiar to anybody with web development experience, regardless of whether they've used a given framework in the past. It also means that the framework is more likely to play nicely with other web technologies such as third-party libraries and frameworks.
- **What past development experience does the team have?** Certain frameworks can

have a steeper or shallower learning curve, depending on the experience of the team.

- **How is the team organized?** Do you need designers and developers to be able to work together on the same project?
- **What kind of commercial and community support do you need?** How important is it to be able to pick up the phone or send an email to the team or company responsible for the framework, what kind of community are you looking to join?

Let's look at where Aurelia sits in terms of each of these questions; in doing so, you'll get a look at the kinds of problems that Aurelia helps you solve, and some of the features available in the Aurelia toolbox.

1.1.1 How complicated are the applications we are trying to build?

With most SPAs, it's helpful to have a minimal set of tools to build the kind of experience that users expect. If these tools are not present in the framework, then you may need to either bring them in as a third-party project dependency or build a bespoke implementation. Aurelia provides a core set of functionalities that most SPAs need out of the box as a set of base modules. Most of these modules are available as optional plugins, so if you don't need a part, you can just leave it out. Here is a basic list of the features that Aurelia offers. I'll include only a brief definition at this point to give you a taste of what's available. We'll dive into each of these topics in much more detail later.

THE BASICS - SPA BREAD AND BUTTER

The following functionality is bread and butter to almost every SPA, regardless of the complexity level:

- **Routing:** Users of an SPA expect your application to behave like a standard website. This means that they should be able to bookmark a URL and get back to it later, and navigate between the different states of your application using the forward and back buttons. The Aurelia Router solves this problem by allowing you to build URL-based routing into the core of your application. Routing also allows you to take advantage of a technique called *Deep Linking*, which allows users to bookmark a URL from deep inside the application (for example a specific product on an ecommerce site) and return to it later.
- **Data-Binding & Templating:** Virtually every SPA needs a way to both take input from the page, either via DOM events or input fields, and push it through to the JavaScript application, and conversely push state changes back to the DOM to provide feedback to the user. Take a contact form example. We'd need a way of knowing when the email field was modified so that we could validate it in our JavaScript application. We'd also need to know what the value of the input field was so that we could determine the validation result. Once we'd validated the input field we'd need to return the result to the user. Data-binding and templating are Aurelia's way of achieving this.

- **HTTP Services:** Most SPAs are not stand-alone; they need to communicate with or get their data from external services. Aurelia provides several options out of the box to make this easy, without the need to pull in any third-party JavaScript libraries like jQuery Ajax.

GETTING MORE ADVANCED - BEYOND BREAD AND BUTTER

As an SPA increases in size and complexity, you run into a new set of problems. When you run into these problems, it's useful to have the tools at hand to solve them.

- **Components:** One of the tools Aurelia provides for dealing with complexity is *Components*. Components are a way of taking a user interface layout and breaking it into small chunks that we can then compose together to build up the entire view of your SPA. In a way, you can think of the components of your page like Objects in a back-end system built using Object Oriented Programming (OOP), such as Ruby, Java, or C#.
- **Inter-Component Communication:** Following the OOP analogy, where in OOP objects can notify each-other of changes in application state, our components also need a way of talking to each-other. Aurelia has several options for how we can implement this kind of behavior. The appropriate option will again depend on the complexity of our application in terms of the number of components and how inter-related they are. We'll dive into inter-component communication in depth in Chapter 4.

Most SPAs start basic and become more complicated over time. Aurelia allows you to reach for tools to deal with the given level of complexity just when you need to, but avoids overloading you with that complexity until you are ready for it. By the end of this book you'll be equipped to handle SPAs with varying complexity levels, and you'll know which tool to reach for from the Aurelia toolbox to deal with each level in the simplest way possible.

1.1.2 How important are web standards compliance to the team?

Imagine you're tasked with building an international website that needs to be accessible by users across the globe, some of whom may be vision-impaired, or have poor quality internet connections. Web standards provide a common base for the web. Devices and browsers are built to web standards specifications, and as such, sticking to this same specification when building websites gives you the best chance of supporting a plethora of devices. The web standards are also focused on accessibility; in fact, there is an entire standards committee called the *Web Content Accessibility Guidelines* (WCAG) that is devoted to it. So, following web standards also gives you a set of tools out of the box to enable support for users with a diverse set of accessibility requirements. A simple example of this is `alt` text on images, which allows screen readers to give vision impaired users a description of the images on your site. At the same time, we reduce our future development costs by building on a stable and well-understood technology set. This makes it easier to bring new team members onto a project who don't necessarily know Aurelia, and allows us to make use of the vast array of

third-party JavaScript and CSS libraries that weren't built with Aurelia in mind, which in turn improves maintainability and reduces development costs.

Wherever possible, Aurelia uses existing browser technology rather than re-inventing the wheel. A simple example of this is HTML markup. Aurelia uses standards-compliant HTML, which allows both humans and screen-readers to read the page source without needing to understand how Aurelia works. As we explore the Aurelia framework, I'll highlight various points where the Aurelia team have leaned on an existing standard web technology to implement a given feature.

1.1.3 What past development experience does the team have?

In the world of web development, the number of new technologies to learn can often seem overwhelming. Given this reality, any boost your team can get in terms of building on past development experience can save you a lot of time. Some of the concepts in Aurelia will feel very familiar to those with OOP experience, such as patterns like Dependency Injection, Model-View-View-Model, *or* Event Aggregator. Don't worry if these concepts are not familiar to you at this point because we'll delve into each of these throughout the course of the book. On the other side, Aurelia should also be easy to pick up for people with a good amount of experience with vanilla JavaScript, HTML, and CSS due to Aurelia's close adherence to web standards.

1.1.4 How is the team organized?

The concept of separation of concerns between your HTML file (which provides the structure of the page), your CSS file (which determines how the page looks), and your JavaScript (which determines how the page behaves has been around in Web development for some time. The idea is that by splitting out these concerns and managing them separately, you should be able to work on any of these independently of the other. It also means that team members with the relevant skill set should be able to work on one piece of the picture without needing to know the in-depth details about the other pieces. For example, a developer should be able to put together the basic HTML structure and JavaScript behavior to then be styled by a team member with more of a focus on User Experience or Design.

Aurelia's opinion on this is that that this concept of *separation of concerns* is no less important in the world of SPA development than it is in a more traditional server-centric web development approach.

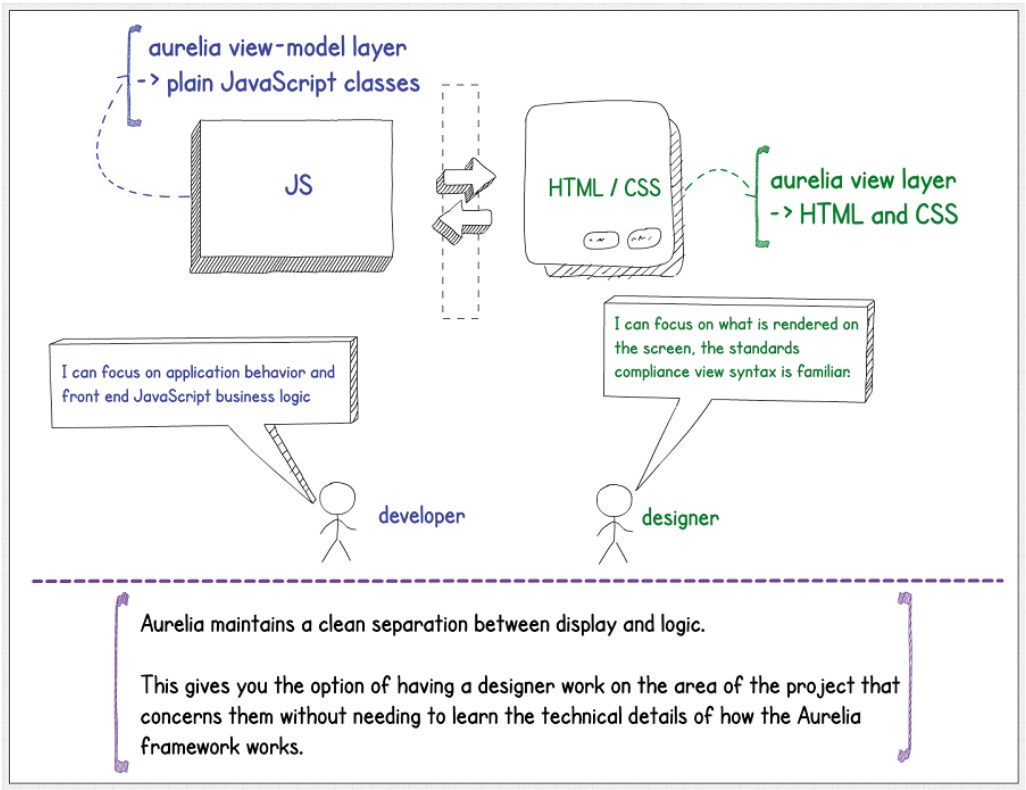

Figure 1.1 Aurelia maintains a separation of concerns between the structure, style, and behavior of pages. This allows for these pieces to be worked on independently, and by the person best suited for the job.

This approach gives you the maximum flexibility. You still have the option of having one person manage an entire vertical slice of the application - JavaScript, HTML, and CSS - but you also have the option of splitting these out if that's the way you'd prefer to work within your team or company.

1.1.5 What kind of commercial and community support do you need?

Besides the technical details of any technology choice, an important aspect to consider is how well the product is supported. The level of support available for a given technology can have an enormous impact on how successful it is within your company. These are some metrics to consider.

LEARNING / TRAINING

How easy is it to take somebody with no experience in the technology and upskill them to the stage that they are productive with it? There are several factors that play into this:

- **Documentation:** Aurelia has a very detailed set of documentation called the Aurelia-Hub. This is actively maintained by the project core team. Often, issues that are raised on GitHub result in a change in the documentation to clarify how a feature should be used.
- **Training:** Aurelia has a training program that makes it possible to receive in-person or online training from an Aurelia expert. This training is official and endorsed by the Aurelia core team. Often, it's even provided by core team members, giving you direct access to people with the most experience working with Aurelia.
- **Community:** If you've not come across it before, Gitter is a chat client like Slack but focused on allowing threaded conversations on open source projects. The Aurelia community has an active Gitter chat room, and often you'll get a detailed answer to any questions you might have.

SUPPORT

Like most popular JavaScript frameworks today, Aurelia is open source. But, unlike most alternatives, Aurelia is one of two SPA frameworks that has commercial support available. Where frameworks such as Angular or React are developed and maintained by Google and Facebook respectively, it's not possible to pay for somebody from these companies to assist you if you get into trouble or just want a little extra guidance on a project. Conversely, BlueSpire Inc - the company behind Aurelia - offers commercial support contracts that can be tailored to the needs of your company.

Support is also available in the standard forms that you'd expect from an open source project. Aurelia core team members are quick to respond to GitHub issues or questions on Gitter.

As somebody from a technical background, it can be easy to overlook the fluffier aspects of choosing a technology like support and training. But, these aspects make a difference when you look at how a technology will be picked up and used by the company long term.

1.2 What kind of projects make Aurelia shine?

To understand what kind of projects Aurelia works best with, let's look at the web development models that are available, and how they might consider the context of a sample application. Imagine that you are tasked with building an ecommerce system. This system consists of a set of the following distinct groups of functionalities:

- **Blog:** News and updates about new products or events. This needs to searchable and is mainly a read-only system.
- **Product List:** A listing of all the products that your company has on offer.

- **Administration:** Administrators need to be able to add new products and view statistics of what users are doing on the site.

SPAs can be structured in several ways, but for our purposes, we can split these into the following four main categories:

1. Server-side application with a sprinkling of JavaScript
2. Client-side-rendered SPA
3. Hybrid SPA
4. Server-side-rendered SPA with client-side continuation.

1.2.1 Server-side application with a sprinkling of JavaScript

Figure 1.2 represents your traditional PHP/JSP/ ASP.NET/ Ruby on Rails-style website. In this model, the user requests the product list, then the server is responsible for rendering it into an HTML file and returning it to the user. Once the page has reached the client side you may have some simple JavaScript for things, such showing a light box image of a product.

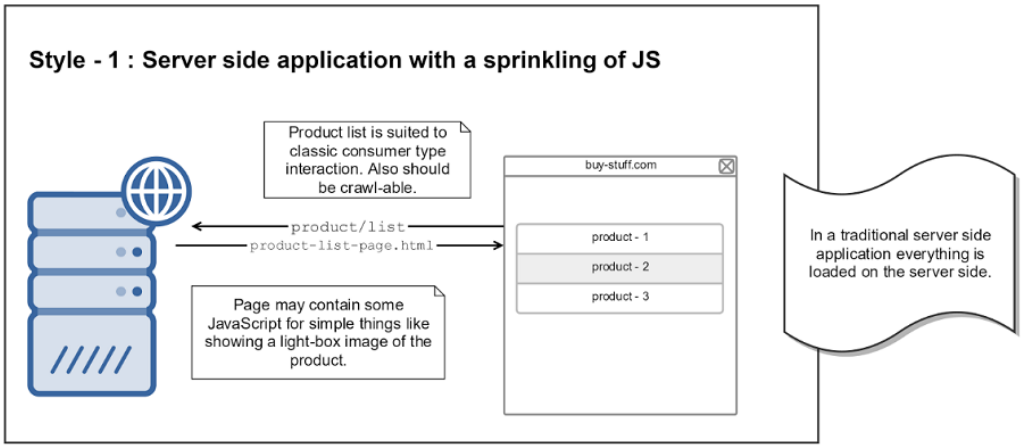

Figure 1.2 Server-side applications retrieve the entire page load as one rendered resource from the server (in addition to any CSS and JS that would typically be loaded separately).

BENEFITS

- The entire page load is rendered on the server, so scaling the server allows us to improve page load times without us needing to be concerned with the performance of the device used the render the page client side.
- Page crawlers can effectively crawl our site for products and blog entries because JavaScript (which is has patchy support by these technologies at best) is not required to render the page.
- Initial page load is generally fast. The user isn't returned our page by the server until

the page is ready to go.

DRAWBACKS

- Though the initial page load time is relatively quick, subsequent interactions to the page also go through the same life-cycle of a full HTTP request to the server and re-render of the entirely new page in the browser. This is ok for the features such as the Blog and Product List but is not as ideal when the user needs to perform a set of interactions on the site and receive rapid feedback (such as commenting on a blog or creating a new product).

1.2.2 Client-side-rendered SPA

In contrast to the server-side application, the entire payload is loaded up front and returned to the browser in a single batch in a client-side SPA. This payload contains the application scripts and templates along with (optionally) some seed data required on the initial page load. In this case, the page is rendered on the client side rather than the server side. Traditionally in this model we'd have the entire e-commerce website returned when the user visited the initial page. Following this, as the user browsed through various pages such as products and blog entries, we'd re-render the page client side, based on Ajax data we'd received back from the server.

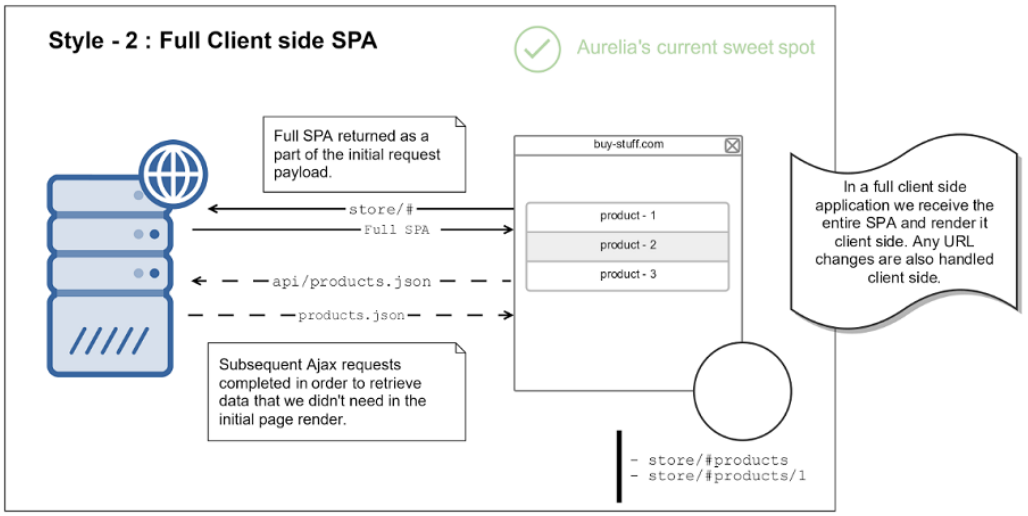

Figure 1.3 Full client-side SPA. Entire application bundled and returned in one or two responses from the server and rendered in the browser. Subsequent Ajax requests completed to populate data not required in the initial page load.

BENEFITS

- As the user clicks around, viewing products and comments on blog threads, the response times are lightning fast. The only time the client needs to talk to the server is when it needs data that the user hasn't seen before. This is generally optional data that may not need to be loaded at all. In these cases, it's generally easy to show a spinner or a similar indicator to give the user instant feedback that the data they want is being fetched, then proceed to show them the re-rendered UI once the data has been received.
- After initial page load, you can also make the page feel more responsive by executing multiple concurrent asynchronous calls to the back-end API. When these asynchronous calls are completed you can provide the results to the user by re-rendering only the impacted DOM fragment. An example of this would be retrieving the list of products as JSON from an HTTP response and re-rendering only the product-list while leaving the rest of the page in-tact.

DRAWBACKS

- **Large response payload:** Because we are loading the entire page in one request, users are forced to wait while the page loads. Typically, a spinner or a similar indicator would also be shown in this case. But with large applications, this can be time-consuming. Because 47% of users expect a website to load in 2 seconds or less, there is a risk that you may lose some of the visitors to your site if this initial application load exceeds this boundary.
- **Unpredictable Page Load Times:** Because we are delivering a big chunk of JavaScript and state to the application, page render times can also be unpredictable. If you're visiting the page from a modern device with a fast processor and high internet bandwidth, there is a good chance you'll have the lightning fast experience we've come to expect from SPAs. However, what if you're in outback Australia attempting to load the application on a 5-year-old smart-phone over a flakey 3G connection? The experience will be altogether different. Conversely, a traditional server-rendered web application is more predictable (at least in terms of the time required to render the application before it is returned to the user). HTTP servers can be more easily updated and scaled to improve these render times, bringing the issue back into something that we can control, rather than being subject to whatever device the user happens to be visiting the page from.
- **The crawler issue:** Some technologies such as web crawlers are not built with the ability to process JavaScript. This is a significant issue if we're attempting to build a public-facing SPA that we want to appear in search engine results.

BEST OF BOTH WORLDS

A good option would be to split the application into several separate sites. The blog might still be done in the traditional server-side rendered style (mainly for SEO purposes). But, you could potentially develop the administrative site and shopping cart as separate micro-sites, providing the user with that rich interaction needed for this style of user interface.

1.2.3 The hybrid approach: server side with SPA islands

Imagine a new set of requirements have come down from management. After great initial success, the company is looking to expand the use of the site and sell direct. As such, the site needs some interactive pieces like a shopping cart and order screen. Taking the hybrid approach, you'd create the following three new routes in the website:

- `site/store/cart`
- `site/store/order`
- `site/admin`

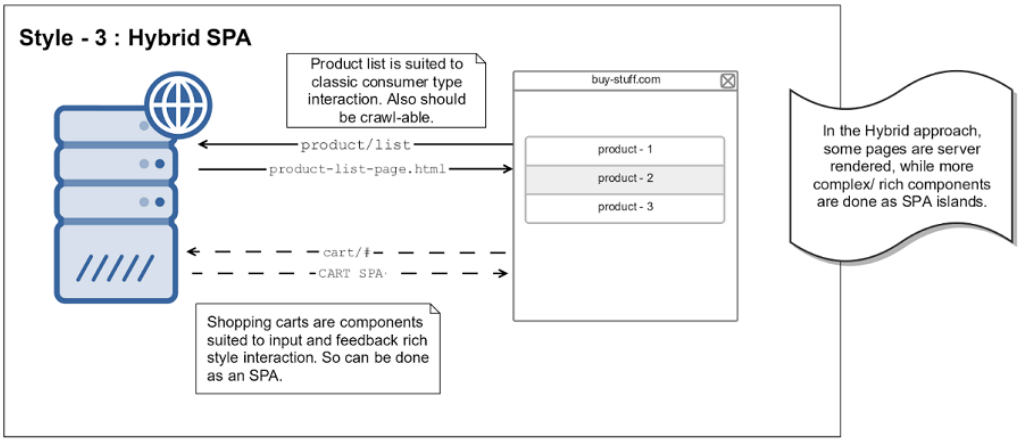

Figure 1.4 Hybrid SPA approach. In this model, the main *content* parts of the site would be done in the traditional server-side approach. The *product-list* and *blog* endpoints would return the relevant HTML rendered from the server. However, we'd then create multiple smaller SPAs to add rich interaction to the parts of the site that need it such as the store.

Each of these new routes would host an entire SPA. This approach avoids the need to immediately re-write the entire site as an SPA, but still gives you the benefit of building the rich interactive parts of the site in a technology designed for that purpose. It also avoids the large payload size to some extent by splitting the SPA into smaller chunks that can be more quickly retrieved from the server and rendered.

BENEFITS

- Resolves the SEO and unpredictable page-load issues while still providing an interactive experience.
- If you've already got a server-side application, you don't need to re-write it as an SPA from the ground up but instead can break out components of the application that are good candidates for SPA-style interaction over time.

DRAWBACKS

- **Added complexity:** Managing application state and page navigation can get tricky when these concerns are dealt with both at the client-side and serve- side level. Each time a new feature is made, there is a cognitive burden of deciding where something would fit best. These styles of application can get messy if the team doesn't establish some clear guidelines up front for how these decisions should be made.

1.2.4 Server-side-rendered SPA

The Aurelia team is working on a technology called *Server Render*. Like the name implies, the application is rendered on the server side before it's returned to the browser. This technology resolves both the *unpredictable page load times* and *crawler* issues. This is not yet ready for production use but goes a long way toward expanding the set of applications that fall into Aurelia's sweet spot. According to Ashley Grant (Aurelia community lead), at the time of writing the Aurelia core team is working on this as a top priority, so I would expect to see movement on this over the next few months.

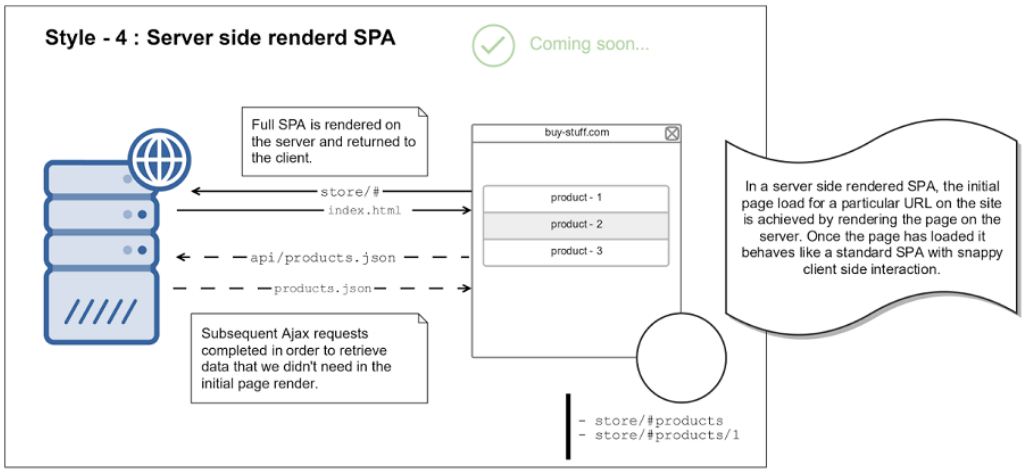

Figure 1.5 In this server-side-rendered SPA model, the initial page render is done on the server side, but once the application has been loaded into the browser, the SPA framework takes over and subsequent interactions are done client side.

BENEFITS

In a way, this model gives us the best of both worlds. The initial render can be done server side, which resolves the issues of unpredictable page-load times and crawler accessibility, but still provides the rich interactive experience that users expect when the page is loaded.

DRAWBACKS

At the time of writing this is still under development for Aurelia. There is also a possibility that this approach will add more complexity to the system architecture by requiring setup of additional components on the server side.

1.2.5 So where does Aurelia sit?

At its heart, Aurelia is designed to manage your entire web application in the style of –client-side rendered-SPA. Any web application where you need a high level of interaction from the user that goes beyond simple content consumption is a good fit for Aurelia. Some example applications that would be a great fit for Aurelia include:

- A messaging client
- A reporting/analytics portal for a website
- A CRUD (create, read, update, delete) application (like the typical invoice/invoice line example)
- An office style application (such as Google Docs)
- An admin portal for a website (like the WordPress admin panel)

What these have in common is a user interaction model that is more like a traditional desktop application than what you'd historically think of as a web site. In our e-commerce example *buy-stuff.com*, a great candidate would be a separate side-kick site to the main website that would be used to handle the administrative operations. You might also use Aurelia to build the shopping cart or blog comment website components as modules of the larger site.

1.2.6 What makes Aurelia different?

With the substantial number of SPA frameworks available today from the heavy hitters like Angular and React to up-and-comers like to Vue and Mithril, it's important to consider what makes Aurelia different. What unique value does Aurelia provide that make it a stand-out choice for building your web applications? I present you with the Aurelia cheat-sheet, four reasons that you can give to your teammates when they ask you this question.

1 – IT GETS OUT OF YOUR WAY

Aurelia applications are built by composing together components built with plain JavaScript and HTML. In contrast, many other MV* frameworks today require a comparatively large

amount of framework specific code in both the view layer and the Model/Controller layer. This increases the concept count, making them more difficult to master and maintain.

2 – CONVENTION OVER CONFIGURATION

Aurelia is developed following the philosophy of *Convention over Configuration* (one of the key reasons why it's clean and simple – see point 1). Convention over Configuration means having reasonable defaults rather than requiring developers to manually specify every option. But what if the convention doesn't suit you? Aurelia makes it easy to override the default conventions when necessary, and we'll go into this in more detail as we proceed through the chapter.

3 – WHEN IT COMES TO WEB STANDARDS, AURELIA'S A PRO

While other framework may pay lip-service to web standards, Aurelia has them at its core, in its bones. Wherever possible Aurelia adopts the standard browser implementation of a feature, rather than creating a framework specific abstraction. A simple example of this is Aurelia's HTML templates, which come directly out of the Web Components Specifications (you can find out more about templates and the Web Components specifications on the Mozilla Developer Network: https://developer.mozilla.org/en/docs/Web/HTML/Element/template).

4 – WHEN IT COMES TO OPEN SOURCE, COMMUNITY IS KING

Aurelia has a thriving open source community. With core team members, and other Aurelia aficionados available on Slack (https://aurelia-js.slack.com/), Gitter (a Slack-like messaging tool for GitHub projects), Stackoverflow, and GitHub help you can always get the help you need to keep up the momentum while building your Aurelia applications.

1.3 Who is this book for?

This book is for any developer wanting to learn how to build rich client-side web applications. If you're a front-end engineer with experience building applications in alternative frameworks such as AngularJS or EmberJS, this book will allow you to put this experience to use with Aurelia. Alternatively, if you've got a good amount of experience engineering server-side web applications with technologies such as ASP.NET, PHP, and have used basic JavaScript for something like form validation or Ajax, then this book will teach you everything you need to build production-ready web applications that work entirely on the client side. Throughout the course of this book, we'll build out an SPA using Aurelia on the front end, with a simple Node.js Express server and MongoDb database on the back end. You should also have some experience with using the command line on Mac or Windows, because we'll be building and running this Aurelia application via Node.js modules. If you do not have any experience building websites that include at least a smattering of JavaScript, then it would be worthwhile getting up to speed with these technologies before circling back to Aurelia.

If you just want to evaluate Aurelia as a potential technology choice for your company, in terms of understanding the community and application architecture, then you'll also get what you need from this book.

1.4 A tour of Aurelia

Often, when you arrive at a hotel in a new city, one of the first things that you are given is a tourist map. Perhaps not a street-level map with the detail of the individual winding roads you need to take, but enough to give you an idea of where the major attractions and suburbs are. With a high-level map in hand, you can generally find where you are going if you get lost by aiming for the attraction you want to see and heading in the right direction. If you need a bit more detail, you might pull out a smartphone to navigate your way through some tricky areas. Similarly, you can think of figure 1.6 as your as your high-level map of the Aurelia framework.

Aurelia applications are built by composing together view, view-model pairs called Components. A view is a standards compliant HTML template, and a view-model is a simple JavaScript class. Binding is used to connect fragments of the view (such as an <input value>) with properties on a view-model. Events raised on the view (such as an input value change) cause a corresponding method to be called on the view-model. Aurelia uses Dependency Injection to construct instances of view-models, providing them with their dependencies at run-time. These dependencies could be service classes (as seen in the diagram, or framework dependencies such as the `@inject` *decorator*.

> **DEFINITION** **Decorators are a new ECHMAScript feature, which at the time of writing are in** Stage 2 of the TC-39's proposal process for ECHMAScript. Fortunately, the Babel.js transpiler allows us to use them today, even though they need to make it to Stage 4 of this proposal process before they are officially adopted as a part of the language. **Decorators allow you to easily and transparently augment the behavior of an object, wrapping it with additional functionality (for example, logging). Decorators are used throughout the Aurelia framework to change the way that objects behave, and can be recognized by their @** `prefix`**. One example of this is the @**`bindable` **decorator** that we imported from the Aurelia framework package. We then declare the books property as `@bindable` **using the imported decorator.**

The Aurelia Router is used to pick the correct component to load for a given application URL.

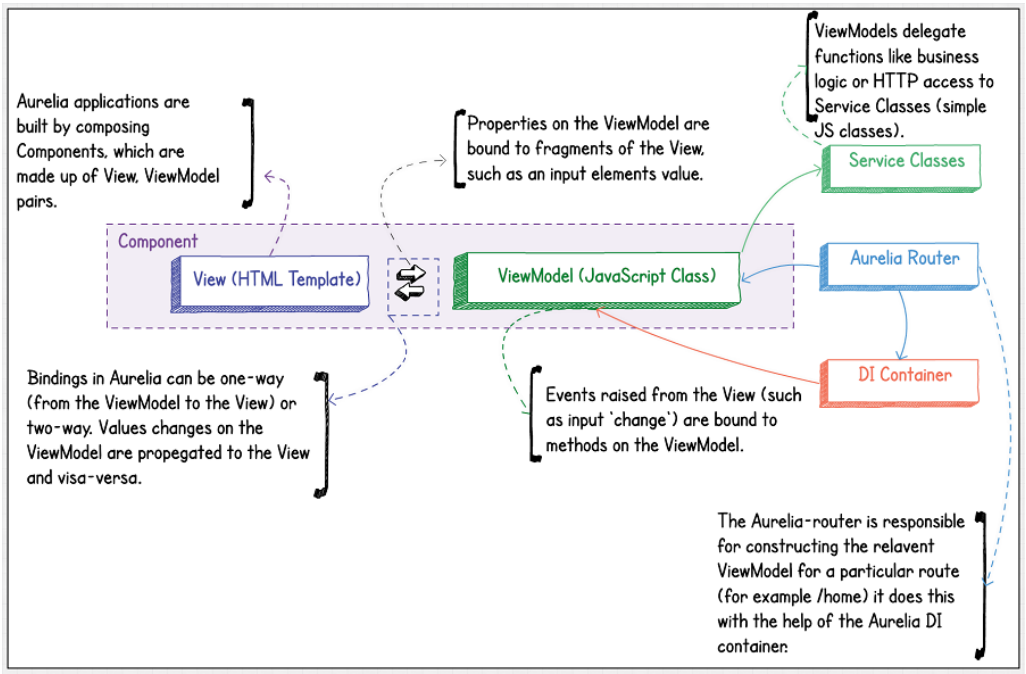

Figure 1.6 A high-level map of the Aurelia framework. At its core, Aurelia is an MV* framework.

There is a lot of detail in the map, but don't worry if you see blocks that you're not familiar with. We'll delve into each area in this diagram to give you a well-rounded understanding of how Aurelia does its thing. Like the concierge, I'm going to draw a line through some of the paths that you'll follow through the framework. These are the code paths that users interacting with the system will trigger every time they load a page or click a button. Like pulling out a smartphone to see more detail about a given location, we'll zoom in on specific parts of the map at points in this virtual tour that represent important aspects of the framework that will be expanded upon throughout the book. There will be a few detours along the way, but by the time I'm finished guiding you through the map, you'll have a much clearer idea of where you are going. Let's get started.

1.4.1 Binding

The core building blocks of Aurelia are view, view-model pairs called components where the view is an HTML template and the view-model is a simple JavaScript class.

> **NOTE** Some components, such as custom attributes and value converters, don't have a corresponding HTML file, but we'll come back to these later.

A technique called binding is used to connect DOM fragments on the view (such as `<input value>` or `<h1>` content) to corresponding properties on the view-model. Changes to properties on the view-model are automatically propagated to the view, causing the relevant fragment of the DOM to be re-rendered with the updated value. Several options are available to control the behavior of how and when the view or view-model are notified of changes in their pair. This binding workflow is illustrated in figure 1.7.

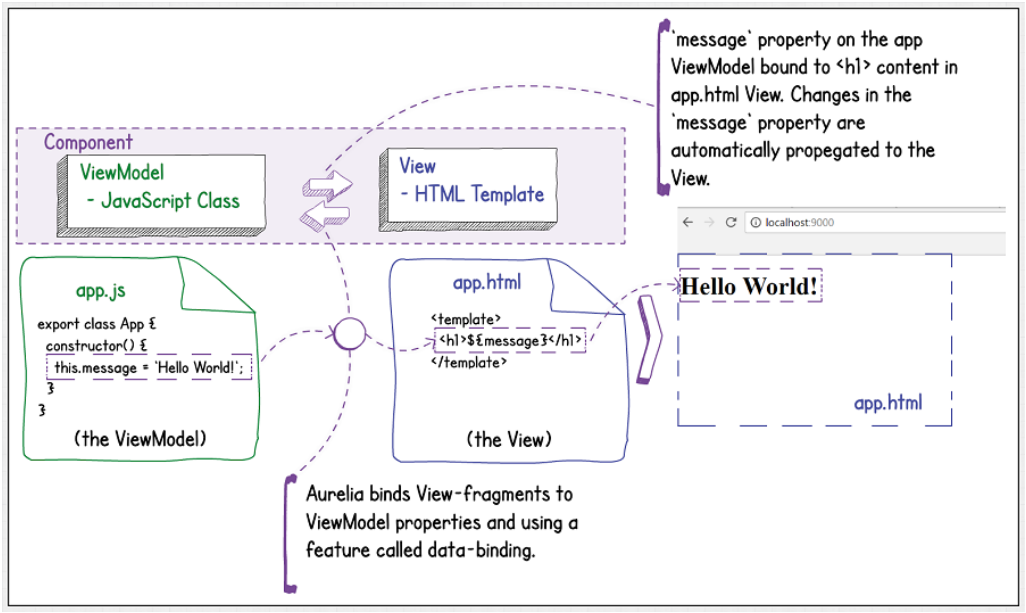

Figure 1.7 The app.js view-model is bound to its pair (the app.html view). When the app view-model is constructed the 'message' property is set to the value 'Hello World!'. This value is bound to the view using data-binding. When this view is rendered the bound value of the message property is rendered to the view, and hence we can see the text 'Hello World!' rendered in the Chrome browser.

1.4.2 Handling DOM Events

The Aurelia binding system is also used to handle DOM events. In this case, any events raised on the view such as `input value change` or `checkbox checked`, or `button clicked` to state a few examples are connected to methods on the corresponding view-model via *binding commands*. Binding commands are Aurelia are commands in the view template used to connect a view event with a view-model method such (`delegate` and `trigger`) – we'll delve into these in detail in chapter 4). As in the case of binding Aurelia provides tools (such as binding behaviors, which you'll also meet in Chapter 3) to control when and why the view-model is notified of events raised from the view. Aurelia's event handling workflow is illustrated in figure 1.8.

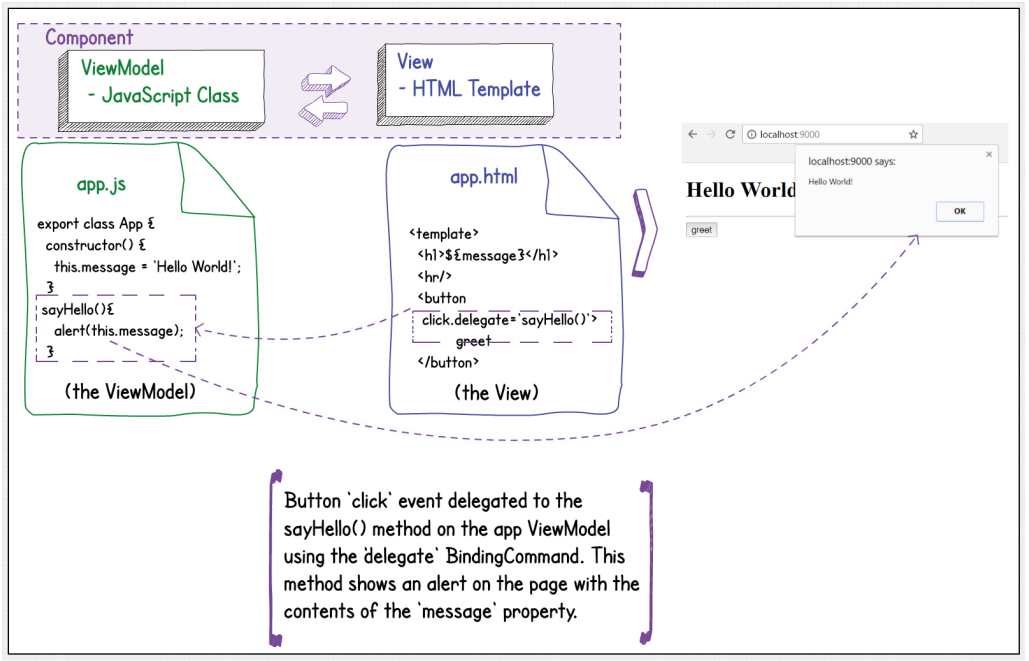

Figure 1.8 The Aurelia Delegate binding command is used to delegate the click event to the sayHello method on the app.js view-model. When the button is clicked, the event is raised, and the sayHello method is called. This results in a greeting alert being presented to the user.

1.4.3 Routing

One of the tools required in most SPAs today is routing. Routing provides you with a way of mapping URLs to application routes. The benefit that this gives you as that you can build an SPA in a way that makes it feel like a real web-site, so conventions such as the back button returning to the previous page work like the user expects. As mentioned, taking advantage of a routing tool in your SPA allows you to implement deep linking, which allows users to visit a path deep within your application (for example `/products?id=1`) and have the page rendered with the state that they expect (in that case the product with the given ID). Aurelia lets us configure an array of URLs called *routes*. When we navigate to a route, it looks up the URL entry in this dictionary and finds the view-model that corresponds to this route. It then constructs that view-model on our behalf. But, how do we get from that constructed JavaScript class to something that's rendered on the page? This question is answered in figure 1.9, which outlines how a typical routing setup in Aurelia works.

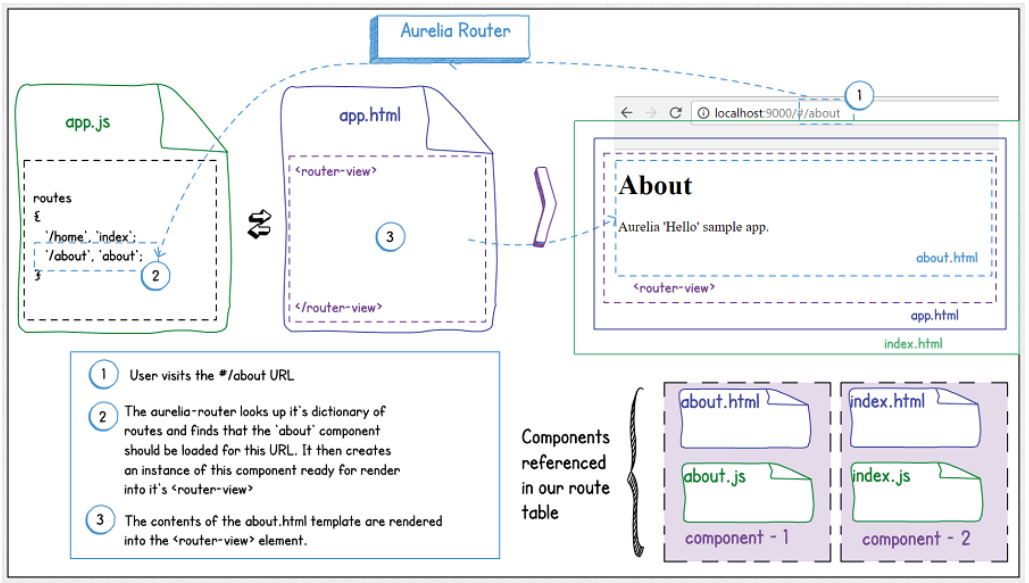

Figure 1.9 Zooming in on the routing component of our Aurelia map. The router works by looking up a URL (in the case of the example the /about URL) and matching it to a route found in the route dictionary. The route dictionary determines which view-model to load. In this case, we've specified that we want the *about* view-model to load whenever a user visits the #/about URL.

In this example, we've added routing to our `app.js` view-model by configuring a set of URLs (home, and about) that correspond to components in our application. In figure 1.9 we've got two components: The `about` component and the `index` component. The Aurelia router takes care of constructing the view-model for the component specified in the route. This component is then rendered inside the `<router-view>` element in the `app.html` view. To get a better understanding of how this process works, bear with me as I take you on a brief diversion to give you some insights into Aurelia's personality. Understanding Aurelia's personality will give you insights into why the framework behaves the way it does in any given scenario. Aurelia is a framework of many strong opinions, held weakly. This means that given any scenario that we might face in our application—in this case, picking the view to show in relation to a view-model —Aurelia has a way that it thinks it should be handled out of the box. But like an open-minded person, Aurelia's opinions are weakly held. If we have a different opinion about the framework, we can tell Aurelia, "No, in this case I want you to pick the view to render based on the Fibonacci sequence," and Aurelia will do that instead.

These opinions are often called *conventions*. Typically, Aurelia's conventions will get you where you need to be about 80% of the time; for the other 20%, you'll need to override these with your own conventions. However, your mileage will vary by how opinionated you are, and how much your opinions diverge from Aurelia's.

One of Aurelia's opinions is that naming is important. Programmers typically put a lot of thought into how we name things. First, it appeals to our sense of organization (sometimes to the despair of those around us). But beyond that, it allows us to remember where everything is in a project and what it does. Naming conventions also allow our team members to make educated guesses about these things and have a fighting chance of being correct when they first start a project. Aurelia has a convention where each view-model file should be named the same as the view file but with a different file extension. For example, a view-model class named `App` should live in a file called `app.js`. The view corresponding to this view-model should then be named `app.html` (figure 1.10).

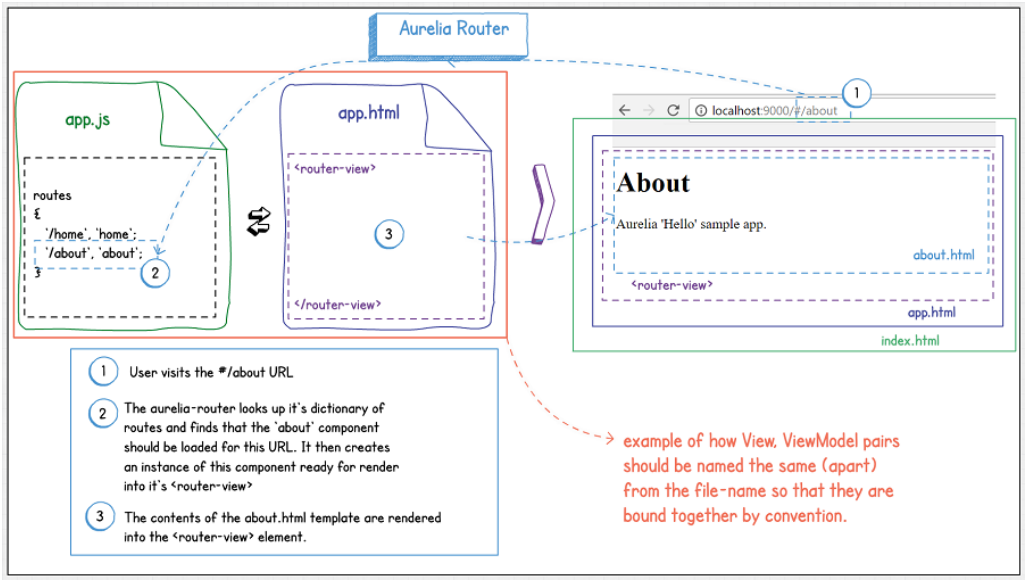

Figure 1.10 Naming the app.js view-model and app.html file in this way allows Aurelia to bind the view and view-model together by convention.

Aurelia has many other opinions, and we'll look at those later, but for now let's return to our dilemma of how we get from the view-model that the router loaded to the view rendered on the page. One option is to define a property in our view-model to hold the path of the view that we want to load when this view model is instantiated. Then when the view-model is loaded, the framework will look up this property, fetch the page, and render it into the DOM. This is a fine approach, and one taken by many frameworks, but not the one taken by Aurelia, which instead uses conventions to create smart defaults regarding which view should be loaded given a specific view-model.

1.4.4 Inside the view-model, and what's this about components?

Now we know that users interact with the Aurelia framework either by navigating to a URL or by interacting with the view, which raises DOM events. In the first case, the appropriate view-model will be loaded for us and instantiated. In the second case, we are already on the page, so our view-model has been instantiated as a part of application startup.

In the example application shown in figure 1.8, we've got only one view-model view pair (the `app.js` and `app.html` files). This pair is called a *component*. A component in Aurelia can represent a section of the UI (for example it could be a nav bar or product list). Components can also be used to encapsulate functionality within your application (for example formatting a date field for display in the view with a value converter). Examples of these kinds of components include value converters, binding behaviors and view engine hooks. We'll cover these kinds of components in Chapter 3. Custom element components are a way of reducing complexity as our application grows by splitting the application into a set of small, well-defined pieces that do one thing well. Those of you from an OOP background can think of this as just another use case of the single responsibility principle.

The first thing that Aurelia does after the view-model has been initialized is to call the view-model constructor. This is the first step in something called the component lifecycle.

1.4.5 The Aurelia component lifecycle

As we interact with the components in our Aurelia application, these components go through a lifecycle—from when they are constructed (for example when we first visit a route that causes a component to be created) and rendered into the DOM, to when we navigate away from this route, causing the component to be cleaned up and removed from the DOM.

Aurelia provides hooks into this lifecycle to allow us to execute behavior relevant at that point in the life of the component. An example of this is the activate life-cycle hook. To hook into when a component's is attached to the DOM we can create an attached method on our component's view-model. You can think of this life-cycle of it like a tour bus, where you let the driver know in advance which attractions you'd be interested in seeing. The driver will then let you know at certain points in the tour that 'We've reached the picnic spot' or 'We've reached the scenic look-out destination.' Then, when you arrive at a destination, you can decide the action you want to take. We'll cover the Aurelia component lifecycle and look at each of the lifecycle hooks available in Chapter 4 when we cover inter-component communication with Aurelia.

1.4.6 Loading data from an API

Suppose that when your page loads you want to retrieve a greeting from a REST API and render it in the app view. To load this data, we can hook into the `created` callback method from the component life-cycle (as shown in figure 1.11), or the `activate` callback from the router life-cycle in the case of a routed component (covered in Chapter 9 – Routing). Zooming in on the data-retrieval area of our map: `app.js` view-model is using an external service

called app-service to retrieve data from a back-end API using the `aurelia-fetch-client`. Figure 1.11 illustrates a typically workflow used to retrieve data from an HTTP API and render it in the DOM.

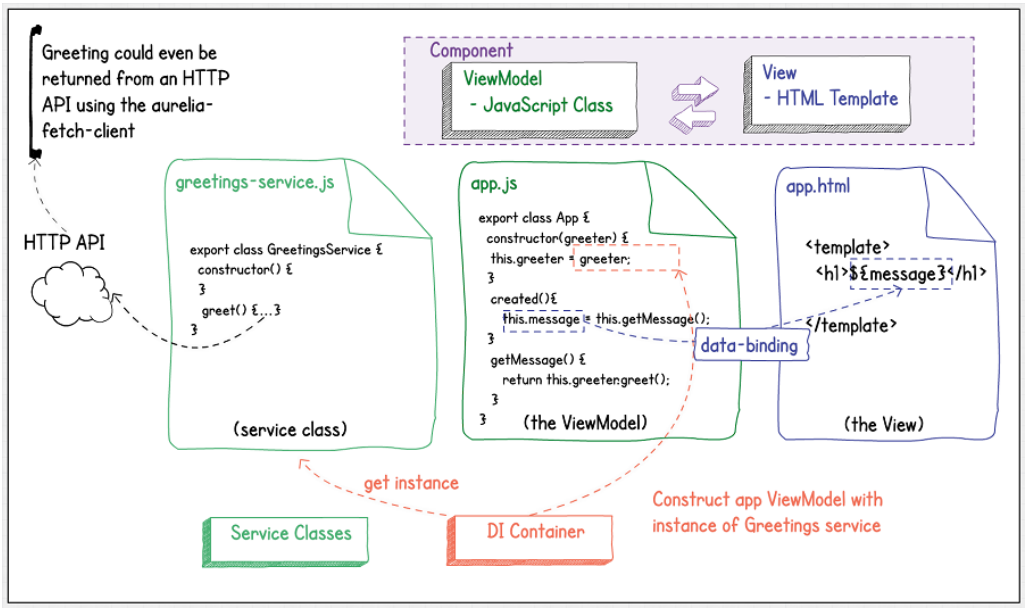

Figure 1.11 The architecture of a sample Hello World application modified to show how a service class could be used to encapsulate functionality such as HTTP API access.

The Aurelia DI container is used to get an instance of the `GreetingsService` class and inject this into the app view-model when it constructs it. This service is then used to retrieve a greeting from an HTTP API. The greeting message is data-bound from the app view-model to the app view, so when our response returns from the HTTP API it's immediately rendered to the DOM.

Having determined where we are going to implement this logic, how do we go about actually doing this? One of the most popular choices for implementing this kind of logic in Aurelia is via the use of a service class (figure 1.11). There is nothing special about a service class. In fact, it's simply a plain old JavaScript object (or POJO). Service classes allow us to separate the concern of retrieving data from an API from the view-model concerns of getting data rendered to the screen. In this case, we would fill out the logic for retrieving the data via Ajax (most likely via the `aurelia-fetch-client`, an HTTP client that makes your life much simpler than the jQuery Ajax `$.get` method, but we'll get into that in more detail later). An instance of this service class can be automatically injected into our view-model using *dependency injection*.

DEFINIITON Traditionally, objects in a system are responsible for managing their own dependencies. This can become challenging as the application grows in scale, increasing the complexity of the relationships between these objects. Dependency injection simplifies this problem by moving the responsibility of creating objects away from the objects themselves and placing it in the hands of the dependency injection (DI) framework. This transition of responsibility is known as *Inversion of Control (IoC)*. Typically, in an application using DI, an object declares which dependencies it requires (often as constructor parameters), and the DI framework provides the relevant implementation of these dependencies at run-time.

1.4.7 Dependency injection with Aurelia

In our simple example view-model we are working with, we have only one basic screen (the `app.html` view), so it will be easy to create a new instance of the class directly in the constructor and do what we need to do. In the real world, however, applications are never that simple. Take Facebook as an example. There is a chat box, a notification widget that tells you how many unread messages you've got, an area that shows you all the ongoing conversations with your friends—and this is only on the chat side of things. If you were to build this in Aurelia, each area would be made up of a set of components. Dependency injection becomes useful when we have multiple components, and we need to manage dependences (such as service classes) between these components. We'll look at how DI simplifies this process in more detail when we explorer inter-component communication in Chapter 4.

1.4.8 Rendering the view

After we've retrieved the data via our service class, we need some way of rendering it back to the screen. One traditional approach that you may be familiar with is to use jQuery. In the world of jQuery, we would have started by pulling our JSON blob into an array of JavaScript objects. We would then have to query the DOM to retrieve the Table element object. After that, we would cycle through each of the values in the array and add these as rows to the table. There are two downsides to this approach.

- **Performance:** It's possible to update the DOM efficiently by carefully replacing only the affected DOM branches that correspond to a change in your JavaScript model even with plain JavaScript or jQuery. But because this optimization step is tedious and time-consuming, many developers skip it and instead replace a larger fragment of the DOM than is necessary. Data binding performs this optimization for you, which makes skipping this step a non-issue.
- **Difference of abstraction Level: In an ideal world, when** writing application code, you should need to concern yourself with only the business problem you are solving. If you are in the flow solving this problem, then need to change gears and think instead about the mechanics of getting the relevant state changes reflected in the DOM it can break you out of this flow. Any break in flow causes a slowdown in development pace and introduces a chance for errors to creep in. Imagine if every time you wanted to

accelerate in the car you needed to think about the process of how the internal combustion engine worked to produce forward motion. A focus on such details might increase the likelihood of accidents. The accelerator pedal is an abstraction that avoids us needing to think about all of this. All you need to do is push down the pedal, and off you go. Aurelia's binding system brings us to a higher level of abstraction, much like the accelerator pedal does when we're driving.

To render the results from our service class call, all we do is save those results into a property that is bound to an element in the view. Aurelia then takes care of it, first notifying itself that a change has taken place, and then applying the relevant changes to the DOM. This keeps us at the same level of abstraction throughout the process, and allows us to focus on the task at hand. It also allows Aurelia to optimize the changes to the DOM. Aurelia has a high-level picture of the changes that need to be made to the DOM. Given this perspective, it's able to find a more optimal way to perform these changes to minimize browser re-render and so on. (We'll go into more about Aurelia's DOM optimizations later in this book.)

We started our virtual tour with user interaction on the page (either a URL navigation event or a DOM event). In the case of the navigation event, we've loaded the appropriate view-model and initiated it via the constructor, which is the first step in its component life-cycle. In the case of the DOM event we've responded to an event triggered in the UI that was bound to a method in the view-model via data-binding. In both cases, we responded to the event by retrieving data from a Web API via a service class that was injected into our view-model via dependency injection. Once we'd received the data back from the service class we saved it into a property value on our view-model that was data bound to a property on the view. This caused that part of the view to re-render and display the list of values to the user.

1.5 Summary

In this chapter, you've learned the following:

- Certain styles of web applications today are difficult to develop using the traditional request/response style of web application architecture.
- Many of these applications (such as admin portals or messaging applications) would have been built in the past as desktop applications.
- The SPA architecture makes it easier to build this style of applications, by providing a set of tools such as data-binding and routing.
- Aurelia is an MV* SPA application framework that provides a similar set of tools to other frameworks such as AngularJS or EmberJS, and is more akin to these frameworks than SPA frameworks such as React which is more of a rendering library.
- Aurelia is a stand out choice in today's web development world due to its focus on Clean Code, simplicity, and Convention over Configuration.
- Aurelia applications are built by composing *Components* which are comprised of view, view-model pairs. Where the view is an HTML template and the view-model is a

JavaScript class.

- Databinding is used to handle events from the view in the view-model, and propagate changes in view-model properties to the view, and changes on the view back to view-models.
- Dependency injection is used to simplify the management of dependencies in Aurelia applications by moving the dependency management responsibility out of the components themselves, and into the Dependency Injection container.
- Aurelia provides routing to allow developers to build SPAs that feel like real websites, supporting the standard web interaction patterns that users are familiar with.

You now have an idea of the kinds of problems that Aurelia solves and at a high level, how it solves them. But if you're anything like me, you're keen to learn how to put this into practice. In the next chapter, you'll get your hands dirty creating an Aurelia application from the ground up. We'll create a virtual book-shelf SPA and start off by adding the ability to add and list books. By the end of this book, we'll have built out a full-fledged SPA with multiple inter-related components, third party libraries (such as Bootstrap and Fontawesome), and the ability to send and receive data from an HTTP REST API built with Node.js, MongoDb and Express.js. This will give you an understanding of the variety of tools that Aurelia offers, and by the time you're finished you'll have everything you need to build your own component-oriented SPA with Aurelia.

2

Building your first Aurelia application

This chapter covers

- Introducing my-books, the virtual book shelf built with Aurelia
- Bootstrapping an Aurelia application with the Aurelia CLI
- Creating components in Aurelia by building view/view-model pairs
- Taking form input using two-way binding
- Responding to view events using binding commands
- Retrieving JSON data from a REST API
- Keeping track of UI state with the Aurelia Router

> **"In theory, there is no difference between theory and practice. But in practice, there is."**
>
> **- Yogi Berra.**

When I first started surfing, I took a lesson to get a jump start on the basics. At the beginning of the lesson, the instructor explained the concept of catching a wave, that idea of springing up from the board at just the right moment to glide down the face. I got it, *conceptually*. Of course, when I attempted to catch the first wave, I face-planted and ended up with a nose full of salt water. My understanding moved from *concept* to *reality*.

2.1 Introducing my-books

If you've ever used an application like https://www.goodreads.com/, you'll be familiar with the concept of a virtual bookshelf. Throughout this book, you'll build an implementation of a

virtual bookshelf called *my-books* to give users the ability to keep track of books that they've read. This application allows me to show you some of the shiny tools in Aurelia's tool belt that make it great for building data-centric SPAs. When you've finished this book, you'll be able to apply what you learn to create solutions beyond my-books. In this chapter, you'll start by creating a simple form to add, and list books. In future chapters, we'll take this further, with animations, charts, communication with 3rd party REST APIs and more.

The grandest of applications start with humble beginnings. The first version of our my-books project is no different. To start with we'll create a single page which allows us to view a list of books, and add books to the list. The layout of the application will look something like figure 2.1. The entire application is hosted inside the `index.html` page. There is also a main application component `app.js`, which includes a `book-list` component to render a list of books to the view.

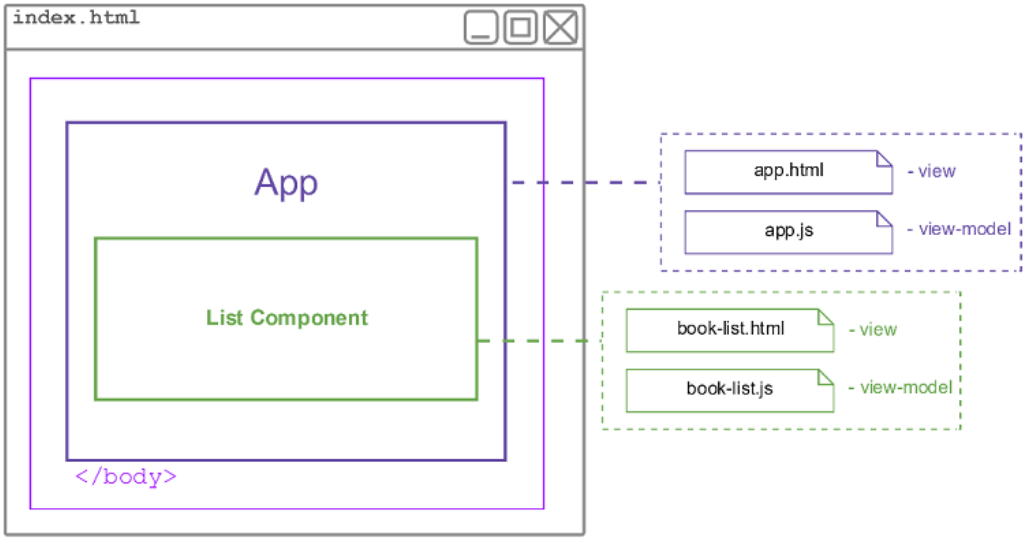

Figure 2.1 my-books application wireframe. We'll build up the my-books application via a set of iterations throughout the book. The first iteration consists of two main components: app, and book-list.

The first step on our journey to creating my-books is to set up the project structure and *build-pipeline* – the process of taking your beautiful ES2015+ modules and getting them running in the browser. For this project, we'll use the *Aurelia CLI* to jump-start our new application.

2.2 Building Aurelia

With Aurelia, one of the themes that may begin to stand out to you is *freedom of choice*. The web development community is varied, both in terms of the kinds of applications we build and

the opinions and approaches we have in mind when building them. Because of this, the Aurelia core team always make sure to include alternative options where available. As such, there are three main options available for developers looking to create a new Aurelia project. The appropriate approach depends on the goal you have in mind for your current project. I'll start by outlining each of the popular choices for creating new projects with Aurelia, and mentioning some of the pros and cons of each. This should assist you in picking the most appropriate option for later projects. We'll then dive into creating a new project via the third option listed - the Aurelia CLI.

2.2.1 Option 1 - Download the quick start

The quick start option is a ZIP package that you can download from the Aurelia website. This is useful if you just want to download the project and get started without performing any kind of setup work installing framework dependencies. Typically, this is not used for real-world applications because it performs transpilation from ESNext to ES5 on the fly in the browser (a slow process). Also, this project is missing some items out of the box, which you'd then need to add anyway. If you'd like to explore this option, you can download the ZIP file from http://aurelia.io/downloads/basic-aurelia-project.zip.

NOTE Aurelia is platform-agnostic but at the moment there is only a ZIP available and no .tar.gz.

2.2.2 Option 2 - Skeleton application

The second option for getting started with Aurelia is to download one of the sample Skeleton applications from GitHub. These are basic shell projects with many of the things that you'll need in a standard single-page application such as routing, a navigation menu, and framework dependencies, already created or referenced. The benefit of this option is that there are Skeletons out there for many of the standard setups you might choose. For example, there is a Skeleton Navigation application that uses Typescript instead of ECMAScript, and one that uses WebPack as an alternative to System JS (these are different module-loading and application-bundling options. Don't worry if some of these terms are fuzzy to you now; we'll unpack them as we proceed through the chapter.

2.2.3 Option 3 - Aurelia CLI

This option is our choice for today. Aurelia applications are built using ESNext or Typescript. Because of this, a collection of Node.js tools are required to create Aurelia applications and perform tasks such as transpilation from ESNext/Typescript to ES5, module loading, and build automation. There are many alternative tooling options for each of these tasks, and the tools of choice within the JavaScript community change rapidly. The Aurelia CLI streamlines the process of creating and building Aurelia applications by setting up reasonable defaults for each of these tools, and in some cases allowing you to swap out a given default for your tool of choice (for example TypeScript rather than Babel.js). The Aurelia CLI is also updated to

incorporate new tooling options as they emerge, allowing you to easily integrate them into your development process. These tools make up a front-end *build pipeline*. When you get to section 2.2.4, you'll see how the Aurelia CLI can be used to make the process of building modern JavaScript applications easier to manage.

> **THE BUILD PIPELINE** You can think of a build pipeline like the standard build process you might be familiar with from the back-end development world. It takes the source code that you write and translates it into something that the browser understands how to run. To learn more about the major components of the modern JavaScript tool chain that we'll use as a part of the Aurelia development and build process.

It's time to get your hands dirty creating you first Aurelia application. In the next section, you'll create the first iteration of my-books using the Aurelia CLI.

2.2.4 Creating the my-books project

Before continuing with this section, please ensure that you have the pre-requisites: Node.js and the Aurelia CLI NPM package installed. Installation instructions can be found in appendix A. Let's begin by generating a new my-books application with the Aurelia CLI.

> **TIP** If you bear with me, I'll get you to run a set of commands at the Aurelia CLI terminal to generate a basic hello world application. That way you'll have a concrete running example to refer to when I describe how this project fits together.

The first step in creating an application using the Aurelia CLI is to run the `new` command to generate the application project structure and configure the build pipeline. Start by running the following command at the terminal/command prompt:

```
au new
```

After running this command, you will be asked to enter the name of your project, as shown in figure 2.2.

Figure 2.2 The au-new command initiates a wizard that will ask you a series of questions required for Aurelia to set up your build pipeline and project structure. The answers to these questions determine which utilities are used at each step of the pipeline.

At the prompt, enter the name of your project – my-books. A new folder with this name will be created to host your Aurelia application as a subdirectory of your current directory.

After entering the project name, select the project setup option you want to use: press enter to select the *Default ESNext* option. This selects the standard build-pipeline configuration. If you want more control over the option for each step of the pipeline, you could select 3 (*Custom*).

```
[aurelia-app]> my-books

Would you like to use the default setup or customize your choices?

1. Default ESNext (Default)
   A basic web-oriented setup with Babel for modern JavaScript development.
2. Default TypeScript
   A basic web-oriented setup with TypeScript for modern JavaScript development.
3. Custom
   Select transpilers, CSS pre-processors and more.

[Default ESNext]>
```

Figure 2.3 The wizard advances you to the next setup options after you enter the project name.

As figure 2.4 shows, you now have the choice for whether you want to proceed with the project setup, given the displayed set of pipeline configuration options or abort and start again. This is useful if you realize at this step that you need to tweak any of the configuration settings before the project is created. If you select *option 3 (abort)* the wizard will exit and you'll need to start from the beginning.

```
[Default ESNext]>

Project Configuration

    Name: my-books
    Platform: Web
    Transpiler: Babel
    Markup Processor: Minimal Minification
    CSS Processor: None
    Unit Test Runner: Karma
    Editor: Visual Studio Code


Would you like to create this project?

1. Yes (Default)
   Creates the project structure based on your selections.
2. Restart
   Restarts the wizard, allowing you to make different selections.
3. Abort
   Aborts the new project wizard.

[Yes]>
```

Figure 2.4 You can see the list of selections you made, with default setup options configured as *Babel*

for our transpiler, the *Karma* test runner, and *Visual Studio Code* for our editor.

Press *enter* to proceed with the *default* option and create the project.

The next step in the project-setup wizard (figure 2.5) determines whether project dependencies (such as RequireJS), should be installed as a part of the project creation process.

```
[Yes]>
Project structure created and configured.

Would you like to install the project dependencies?

1. Yes (Default)
   Installs all server, client and tooling dependencies needed to build the project.
2. No
   Completes the new project wizard without installing dependencies.

[Yes]>
```

Figure 2.5 You can select Aurelia project dependencies in the wizard—for example, the NPM package required to build and run your project. At this step, you can decide whether it should go ahead and install the dependencies or if you would like to do it later.

Installing the dependences should take a few minutes, depending on your internet connection speed. Press the *enter* key to go ahead with the default option.

After the project set up has completed, you should see the following output. If you run into any errors at this point, the best option is to start from scratch with a new project directory. It's also a good idea to make sure that your version of NPM and the Aurelia CLI are up to date to avoid any version mismatch related issues.

UPDATING NPM AND AURELIA You can upgrade to the latest version of NPM using the command `npm install npm@latest -g`. You can get the latest version of the Aurelia CLI via NPM using the command `npm update aurelia-cli -g`.

When you complete the final step in the project setup wizard (figure 2.6) there are several other *Aurelia CLI* commands available now that you have the default project structure in place. You can see a full list of the Aurelia commands by running the `au help` command.

```
  +-- lodash.isequal@4.5.0
  +-- merge-stream@1.0.1
  +-- strip-bom@2.0.0
  | `-- is-utf8@0.2.1
  +-- strip-bom-stream@1.0.0
  | `-- first-chunk-stream@1.0.0
  +-- through2-filter@2.0.0
  `-- vali-date@1.0.0

Congratulations! Your Project "my-books" Has Been Created!


Now it's time for you to get started. It's easy. First, change directory into your new project's folder. You can use cd
my-books to get there. Once in your project folder, simply run your new app with au run. Your app will run
fully bundled. If you would like to have it auto-refresh whenever you make changes to your HTML, JavaScript or CSS, simply use the
--watch flag. If you want to build your app for production, run au build --env prod. That's just about all
there is to it. If you need help, simply run au help.


Happy Coding!
```

Figure 2.6 After completing the final step of the wizard, you're shown some of the other commands that the *Aurelia CLI* provides such as *au build* which you may want to run now that you have the default project structure in place.

Your new Aurelia application includes the project structure and dependencies. We'll delve into this project structure in detail momentarily, but first, to verify that everything has been set up correctly, let's run the default project to see our obligatory hello world page in the browser.

In addition to generating the project structure, Aurelia CLI commands can be used to execute the build-pipeline steps necessary to transpile and package your code, and then get your project running on a simple HTTP server.

> **DEFINITION** Transpilation in the context of Aurelia is the process of taking our source files (ESNext or Typescript) and translating them to the ES5 format supported by today's browsers. This is one of the steps in the *Aurelia Build Pipeline* discussed in Appendix B, and is implemented using a Gulp task run by the Aurelia CLI when you execute the au run command.

To launch the my-books application, change directory to the my-books subdirectory and run the `au run` command at your terminal / command prompt:

```
cd my-books
au run
```

After executing the `au run` command, the my-books application should be hosted in a simple HTTP server on port 9000. Open a browser and navigate to http://localhost:9000 to view the default *Hello World* page. Figure 2.7 demonstrates the page that you should see in the browser.

> **NOTE** For best results, use the current version of Google Chrome, Mozilla Firefox, Microsoft Edge, or Apple Safari. Older browsers such as IE9 are also supported by require polyfills. You can find more about running Aurelia in older browsers on the Aurelia Hub website http://aurelia.io/hub.html#/doc/article/aurelia/framework/latest/app-configuration-and-startup/3.

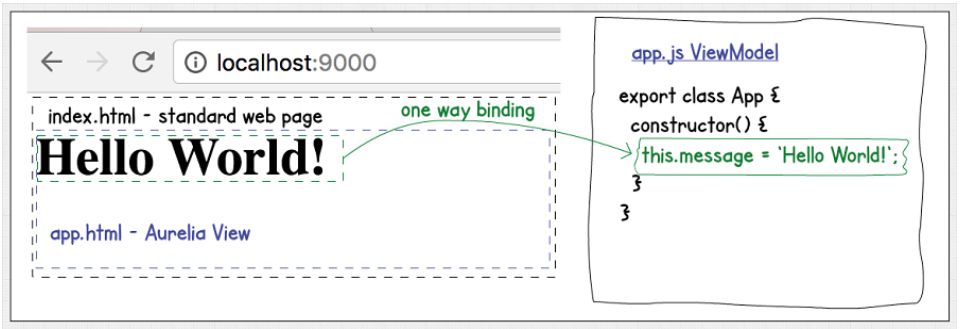

Figure 2.7 Default page shown when setting up a project via the au-new command. This is useful as a smoke-test to ensure that everything has been shown correctly. The default application consists of an app.html view and app.js view-model. One-way binding is used to render a message from the view-model to the view.

Troubleshooting

If you experience this error when using the `au run` command—`SyntaxError: Block-scoped declarations (let, const, function, class) not yet supported outside strict mode`—there is a chance that you are running an unsupported version of Node.js. At the time of writing, this error can be resolved by upgrading to Node.js LTS. It's also worthwhile checking the Aurelia documentation at the following locations for details on the current pre-requisites: http://aurelia.io/hub.html#/doc/article/aurelia/framework/latest/the-aurelia-cli/1.http://aurelia.io/hub.html#/doc/article/aurelia/framework/latest/the-aurelia-cli/1.

Even though you'll remove this default page as you continue creating your virtual bookshelf, this page is useful as a smoke test to ensure that everything has been configured correctly.

> **NOTE** There are so many moving parts in the front-end build-pipeline that it can be difficult to keep track of them all. The project creation wizard has given us an effective shortcut, allowing us to set up all the required build-pipeline plumbing in minutes rather than hours. Additionally, because we're now using a pipeline configured by the Aurelia core team, we can lean on their experience. You can read through the source code of some of the gulp build tasks to familiarize yourself with the latest techniques and practices.

2.3 One-way and two-way data-binding and event delegation

In the preceding section, you generated the my-books project structure and ran the default application using the `au run` command. Executing this command hosted the website inside of a simple HTTP server and rendered a basic *Hello World* page on http://localhost:9000 in the browser. The website is looking basic now though. It's not much use having a virtual bookshelf that is only capable of greeting users. What users really want to be able to do is manage their books. The initial layout of the my-books application will look something like figure 2.8.

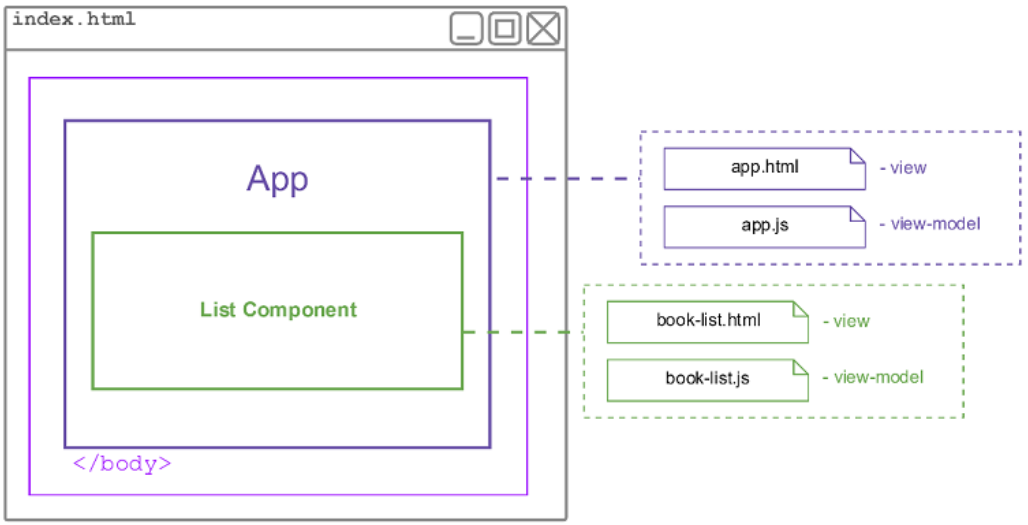

Figure 2.8 This is the my-books layout that we are aiming for in the first version of our application. It has one sub-component (book-list) inside the main app shell, which shows a list of all the books that the user has added. This component consists of a view view-model pair.

Knowing what granularity to break down your application into can be hard. To avoid this problem up front, I often start with the high-level application in-lined on one page, then break this page down into different components as they have enough functionality to justify it. This is the approach we'll take now.

2.3.1 Adding the functionality inline

We'll start by modifying the `app.js` file to include all the desired behavior as shown in code listing 2.1. We'll create the `addBook` method. This method adds a new book to the list of books with the name stored in the `bookTitle` view-model property, clears the current value of `bookTitle` so that the next value can be added, and logs the list of books to the console so that you can view the results. You can get started by opening the newly generated my-books project in your text editor of choice.

> **TIP** My text-editor preference is vscode https://code.visualstudio.com. It has great support for JavaScript, and you can even get plugins for auto-completion of Aurelia classes and HTML snippets.

After we have this simplistic version running, we can start breaking it down into components to make the code cleaner and easier to reason about. Start by creating a new file at `src/app.js`, and then modify the file as shown in listing 2.1.

Listing 2.1 Initial implementation of the app view-model – app.js

```
export class App {

   constructor(){
    this.books = [];      (1)
    this.bookTitle = ""; (2)
  }

  addBook () {            (3)
    this.books.push({title : this.bookTitle}); (4)
    this.bookTitle = ""; (5)
    console.log("Book List ", this.books);     (6)
  }

}
```

(1) Array of books to be bound from app view
(2) Book title to be bound from app view
(3) Method to be called from view to add a book
(4) Add a new book to the array with the bound title
(5) Reset the title
(6) Log the updated book array

The second step is to modify the `src/app.html` view to include an input form and make use of the extra functionality exposed by view-model. Code listing 2.2 shows the source code for the modified version of our app.html file to include a rudimentary implementation of adding and listing books.

Listing 2.2 Initial implementation of app view – app.html

```
<template>
  <h1>Add Book</h1>

  <form submit.trigger="addBook()"> (1)

    <label for="book-title"></label>

    <input value.bind="bookTitle"   (2)
        id="book-title"
        type="text"
        placeholder="book title...">

    <input type="submit" value="add" >

  </form>
</template>
```

(1) Trigger the addBook method on form submit
(2) Bind bookTitle view-model property to the input value

As we saw in chapter 1, because the `app` view and view-model have the same name, Aurelia automatically binds them together by convention. As result of this, submit event on the form

is delegated to the `addBook` method on the `app` view-model using the `submit.trigger` binding command. This wires up Aurelia to capture the submit event and call the `addBook` method whenever the event is fired.

The `value` attribute of the input field is configured for two-way binding with the `bookTitle` property in the value using the `value.bind` syntax. When the value is changed in the book title input field the corresponding value is updated in the view-model, and visa-versa.

The value input on the form that you've added to this view is bound to the value of the `bookTitle` view-model property using the `value.bind` binding command. This binding is two-way, meaning that changes to the value on the view-model are sent to the view and values typed into the input box in the view are send back to the view-model. At the beginning of the form, the form submit event is wired up to trigger the `addBook` view-model method using the `submit.trigger` binding command.

> **TIP** One thing that caught me out a few times when I started using this syntax was that the brackets need to be included in the method name that you are binding to (`addBook()` rather than `addBook`). If you do the latter, you'll be scratching your head wondering why the method isn't being called, so it's good to remember that from the start. This syntax is required because we are calling a function on the view-model rather than reading the value of a property.

This gives you a taste of some of the user interactions that you can set up using the related concepts of event delegation and data binding. In this case, we've set up the event delegation on the `submit` DOM event but this could also be used to bind to any event raised by the DOM, allowing you to handle a plethora of different scenarios. Likewise, we've set up a two-way binding to an input element as a starting point, but this concept can be applied to any DOM element that takes input. A key point to note here is that even though we've not specified the input binding as two-way, Aurelia has automatically created a two-way binding by convention because the binding command is applied to an input element. The logic behind this convention is that we likely want to send input back to the view-model. We'll elaborate on the various scenarios supported by data binding and event delegation in chapter 3 when we take a deep dive into the Aurelia binding and templating engine.

You can view the first iteration of the add-book form by running the `au run --watch` command and navigating to http://localhost:9000 in the browser.

> **TIP** Running the `au run` command with the –watch parameter will spin up the simple web server as before and watch for any file changes, which is useful as you modify the component to add form elements.

Type a value into the input box and click the Add button or press *enter*. The value of the book-list array updates via two-way data binding and is logged to the console. This allows you to verify that your first event delegation trigger and two-way data binding are working as expected.

You should now see the add-book form rendered on the page: when you add a book to the list, it should be logged to the console as shown in figure 2.9.

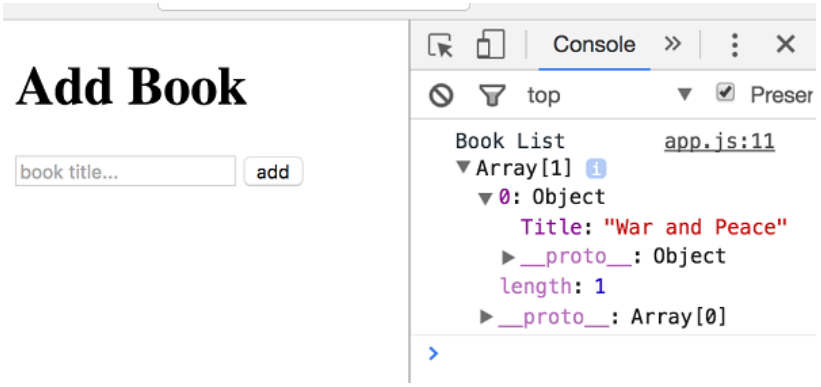

Figure 2.9 This is what the first iteration of your add-book form looks like in the browser.

Now that you can add books to the list, we need a way to view them on the page. We'll do this by rendering the list of books to an unordered list on in the `app.html` view using the `repeat.for="item of array" repeater`. You can think of repeaters like an ES2015 `for…of` loop because they have the same semantics. They allow you to loop through the values of an array and inject content for each item into the DOM. This is often used in combination with the one-way data-binding string interpolation syntax that we saw earlier (`${viewModelProperty}`) to render the value of the item in the current iteration. We'll cover repeaters in more depth in chapter 3. Code listing 2.3 shows the source code changes required to render an un-ordered list of books to the `app` view. Modify the `src/app.html` view to display a list of the books that have been added to the list.

Listing 2.3 Modified app view to include books repeater – app.html

```
<template>

  <h1>Add Book</h1>

  …
  <hr/>

  <ul>
   <li repeat.for="book of books"> ${book.title} </li> (1)
  </ul>

</template>
```

(1) The repeat.for="book of books" syntax can be used to iterate over each of the items in an array.

Within the context of the `<li>` element (including attributes on the element as well as inside the element itself) we have access to the book in the current iteration of the loop. We then use a one-way binding expression `${book.title}` to render the `Title` property of the book in the current iteration to the page. Conceptually, Aurelia converts `<li repeat.for="book of books">` to `<template repeat.for="book of books">…</template>` but this is abstracted to create a cleaner syntax for development time.

If you're still following along, run the `au run --watch` command at the terminal/ command line and navigate in the browser to http://localhost:9000. You should now see the un-ordered list of all the books rendered to the page as shown in figure 2.10.

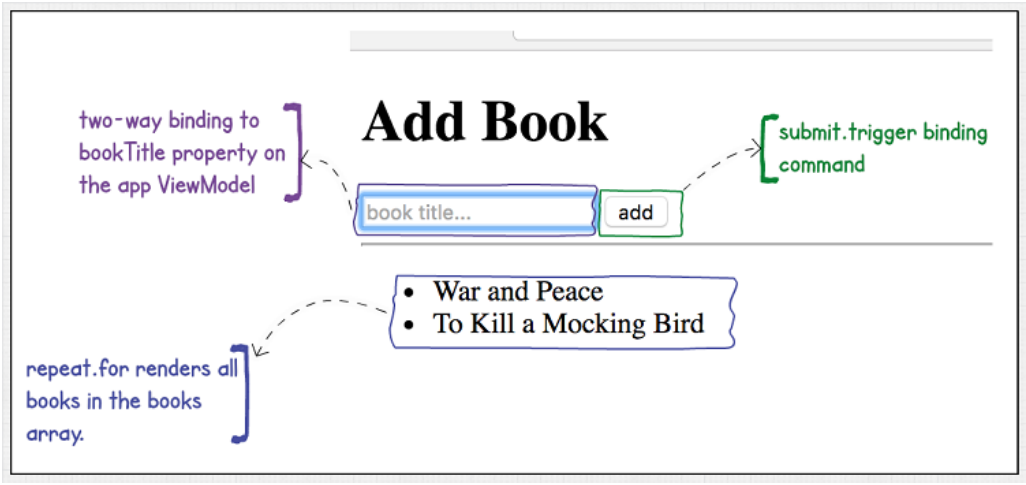

Figure 2.10 Adding books triggers the one-way binding expression from the view-model to the view for the book-list array to be updated which causes the updated list to be rendered to the page.

Let's recap what you've learned so far. You've now used a combination of one-way data-binding, two-way data-binding, iterator expressions, and event delegation to take input from a form, add that input to a list, and render the updated values to the page. A key point here is that you didn't need to do anything special to refresh the list of books after the value was added. The `repeat.for` binding expression allows you to render a collection of items (such as an array, range, or map) from the view-model to the view. Changes on the collection are observed, meaning that additions or deletions from the collection are propagated to the view. We'll cover repeater semantics in detail in chapter 3. This binding system allows you to implement an interaction model where users receive instant feedback based on actions they take on the page, without needing to individually wire up at which points the page needs to be refreshed.

Now that you've completed the first iteration of this page with all functionality developed inline as a part of the `app` component, it's time to look at breaking down the page to extract

the first logical sub-component. In the next section, we'll break out the book `book-list` component, and in doing so will take our first look at Aurelia's custom element feature.

EXERCISE 2.1 – BINDING

We've started the implementation of a simple Aurelia greeter application on GistRun https://gist.run/?id=3c8d0b42001c1cf772b4327af9af3c85. As an exercise, try modifying the application to change the greeting message to the contents of the input box on form submit. You can find the solution here https://gist.run/?id=424ab0c86b196c3a8203b9490a15a6cc.

2.4 Creating components using Aurelia custom elements

At the beginning of section 2.3, we outlined a high-level blueprint of how the my-books page should be composed. The first step in implementing this view was adding the `add` and `book-list` functionality inline.

Now that we have the basic functionality in place, it's time to break out the first logical sub-component to match our original blue-print, as shown in figure 2.11. At the root level of the application there is the app component, which serves as the application shell. There is one sub-component under the app component called `book-list`, which renders the list of books to the page.

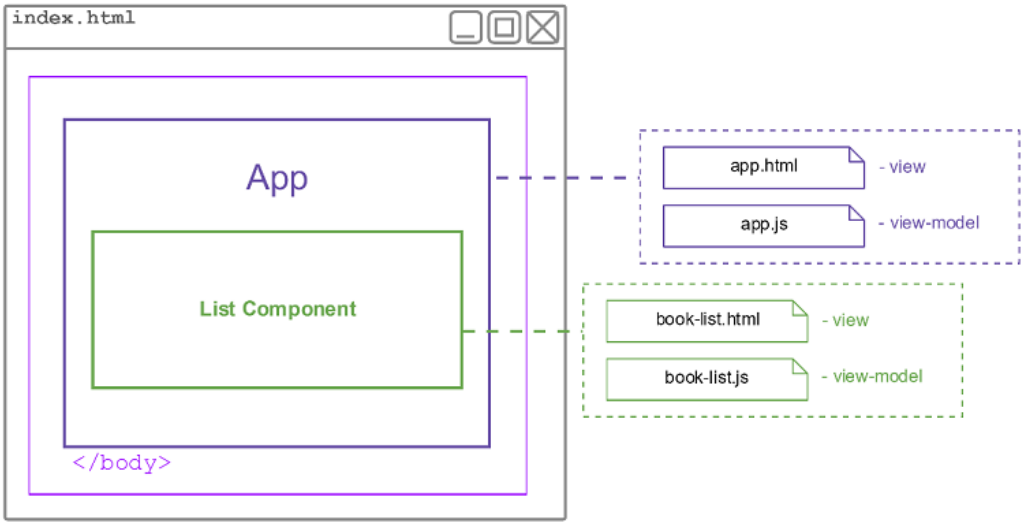

Figure 2.11 The high-level blueprint of the first page of the my-books application shows that the page should be broken down into two components: The app (application shell and add-book form), and book-list components (the rendered list of all books added).

The new `book-list` component will be created as an Aurelia custom element. Custom elements are a way of extending the DOM by creating new HTML elements that can behave like regular DOM elements. To create custom elements, we'll do the following:

- Create a new JavaScript class (the view-model)
- Create a corresponding HTML template (the view)
- Name the view and the view-model the same apart from the extension so that Aurelia can automatically bind them together.

Beginning at the top, let's start by creating the `book-list` custom element by defining two new files (`src/book-list.js`, and `src/book-list.html`) as shown in listings 2.4 and 2.5. This is not technically the cleanest place to put these (imagine what it would look like with thousands of files at this root directory), but it keeps things simpler for now. This view-model will be bare-bones to start off with, exposing only a single `@bindable` property books to be bound from the parent element.

Listing 2.4 Initial implementation of book-list view-model – book-list.js

```
import {bindable} from 'aurelia-framework'; (1)

export class BookList {
  @bindable books; (2)
}
```

(1) Import the bindable attribute from the aurelia-framework package
(2) Define a new bindable property books on the view-model

In listing 2.4, we begin by importing the @bindable decorator. This is then used to set up a one-way data-binding between the app parent component and book-list child component.

> **REMINDER** Decorators allow you to easily and transparently augment the behavior of an object, wrapping it with additional functionality (for example, logging). Decorators are used throughout the Aurelia framework to change the way that objects behave and can be recognized by their @ prefix. One example of this is the @bindable decorator that we imported from the Aurelia framework package. We then declare the books property as @bindable using the imported decorator.

Next, we need to create the `src/book-list.html` view which corresponds to this view-model to complete the new component. Create a new file called `src/book-list.html` and add the code shown in listing 2.5.

Listing 2.5 Initial implementation of book-list view – book-list.html

```
<template>
    <ul>
        <li repeat.for="book of books"> ${book.title} </li> (1)
    </ul>
</template>
```

❶ Refactor the repeater into a new book-list view.

What we've achieved in listing 2.5 is really a bit of housekeeping, cleaning up the project structure, and moving the book-list repeater from the `app.html` into the `book-list.html` view.

Now that we've encapsulated the book list functionality into its own view (`book-list`), we'll need to remove this behavior from the `app` view. You can think of it like changing the level of abstraction in the app view. We've delegated the book addition behavior to child component (`book-list`), so it no longer belongs in the app view. This lifts the level of abstraction in the app component by delegating the details of how books should be added *down* the component hierarchy.

The example in listing 2.6 contains some elements you've not seen yet. The `<require>` element is a custom element provided by the Aurelia framework that behaves similarly to a `require()` statement used to import a module in JavaScript or an `import` statement in ES6. This imports the references you want to use in your view. The `<book-list>` custom element is your new `BookList` component. Wherever you include a reference to this you'll get a copy of it injected into the DOM. If you reference an element without first importing it with `<require>` nothing will be rendered on the page, as Aurelia doesn't know how to interpret it. Modify the `app` view `/src/app.html` to use the newly created custom element, as shown in listing 2.6.

Listing 2.6 Modified app view refactored to use custom elements – app.html

```
<template>
    <require from="./book-list"></require> ❶
    <h1>Books</h1>
    <form submit.trigger="addBook()">
        <label for="book-title"></label>

        <input value.bind="bookTitle"
            id="book-title"
            type="text"
            placeholder="book title...">

        <input type="submit" value="add">
    </form>
    <hr/>
    <book-list books.bind="books"></book-list>  ❷
</template>
```

❶ Import the book-list custom element
❷ Add a book-list element to the view

By using the `<require from="">` custom element, we declare that the `app.html` file depends on the `book-list` module. This indicates to the framework that the module loader that we've got configured (in our case. RequireJS) should import that resource at runtime.

The `books.bind="books"` expression is a one-way data binding. It binds the `books` property of the parent view-model `app` to the same property on the child view model. That way, any books added to the list in the `app` view-model are automatically propagated to the `book-list` view-model. Figure 2.12 shows the data-flow between these components.

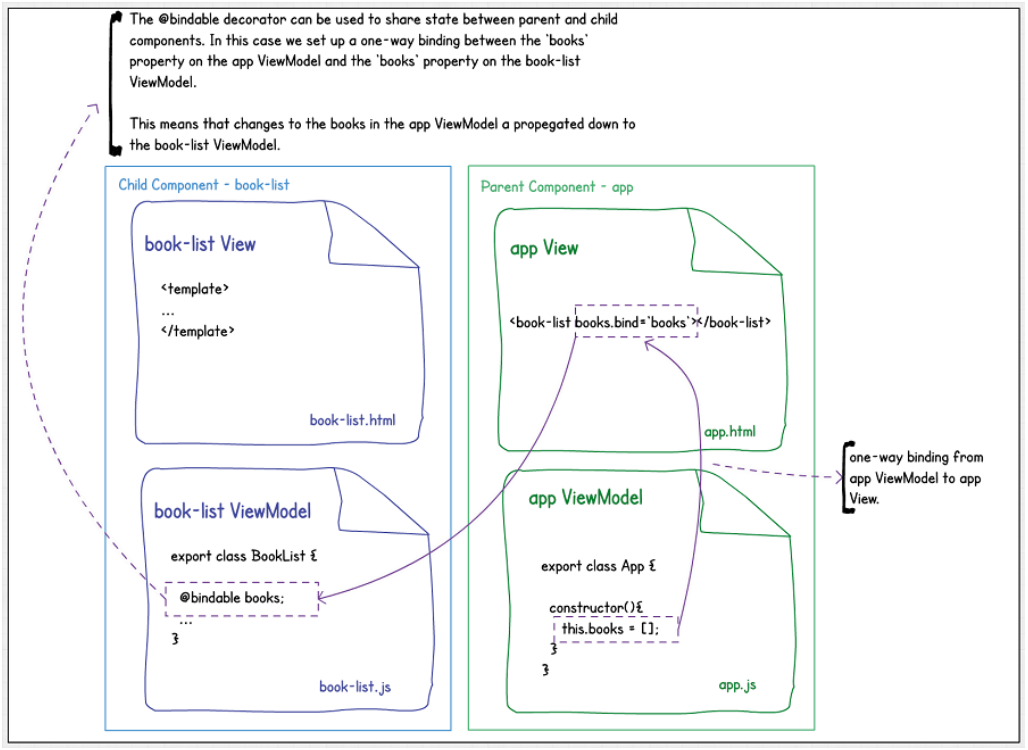

Figure 2.12 As books are added to the books[] array in the app view-model, the changes are propagated down the component hierarchy using one-way data binding.

This pattern that we've followed for breaking down the user interface into a set of custom elements is one of the key patterns you'll follow when creating Aurelia applications. As the user interfaces grow in complexity, it's key to modularize the layout by breaking it down into small, well-defined pieces. The benefit is that you can then take these custom elements and either move them to different areas of your application to adjust to layout requirements changes, or even re-use the components in different contexts. For example, imagine that you want to add the ability for users to upload a book cover image to be stored against the book and displayed in the book list. Further, you also want to allow users to upload an Excel spreadsheet listing all their books to bulk load their entire library. If you build this upload functionality as a custom element component, you can then re-use this same element as-is to

implement both of those requirements. The `book-list` component modularization you just completed would also be useful in this case; to add the ability to render the book cover image for each item in the list, the only component that you'd need to modify is the `book-list` component. This keeps the change encapsulated, making it easier to easier to reason about and test.

In this first iteration of the my-books application, you used custom elements to break down the Aurelia application layout into a set of modular components and linked these components together with data-binding.

> **DISCLAIMER**: There is a fine line to walk when componentizing your application. You want to make sure that creating a given component makes the application design simpler. Although decomposing the application into smaller units can make it easier to reason about each unit, it also adds complexity as you need to think about how these units communicate.

You'll get a better feel of how to decide when to split out a new component throughout the course of the book as we expand on the my-books application. Currently, you're rendering a set of hard-coded books into the `book-list`. Next, you'll make this list more dynamic by loading from a REST API using the `aurelia-http-client`.

> **THINGS TO COME** In chapter 6, you'll work with more detailed view composition and in the process, you'll add the ability to mark books as read and provide a star rating.

EXERCISE 2.2 – CUSTOM ELEMENTS

As an exercise, try modifying the greeter in this Gist to encapsulate the greeting component in a new custom element https://gist.run/?id=5d8b3973b5d752c0d2c79e817355b24a. You can find the solution here https://gist.run/?id=1259c38d3c4927d6842e4624bc6e0907.

2.5 Building services and retrieving data from a REST API

I have a feeling that when you saw the `view-model` source code, you got the gist of what it was doing before you read the explanations. One reason for this is that in keeping with *The Aurelia Way*, the framework introduces minimal extraneous code. For example, there is nothing telling the `book-list` view-model where to find the view template. All you need to do is name it `book-list.html` and the framework figures it out for you. This allows you to keep your classes as close to vanilla JavaScript as possible. The only item that may have caused a blip in your brain while you were reading through the view-model implementations is the `@bindable` attribute that you used to enlist a property for two-way data-binding on the `book-list` view-model. Apart from that, they are *just* ESNext classes. Wouldn't it be nice if it were possible to implement data access in the same way?

In Aurelia, there is no specific concept for service/data-access logic. All you need to do is create a class that provides the data or functionality required by your view-model. This class is then injected by the Aurelia Dependency Injection system.

Next, you'll add a back end to my-books to allow your saved books to be loaded on startup. You'll do this by adding a new class responsible for retrieving a list of books from the back-end API. But first, the setup. The following steps are required to retrieve data from our API:

- Create the my-book JSON seed file `books.json` to stand in as our REST API endpoint.
- Install the `aurelia-fetch-client` using the `au install` CLI command.
- Create the `BookApi` service class to implement the HTTP logic.
- Update the book-list component to retrieve the seed data from `books.json` via the `BookApi` service.

To keep things simple and avoid creating a back-end REST API for now, the current implementation of the my-books API is a JSON file. The REST API will return just the contents of a flat JSON file that will be read from file by the HTTP server Create a new file named `books.json` at the root of your application (at the same level as the `index.html` file called `books.json`, as shown in figure 2.13. Having the file at the root level ensures that the file is served by the HTTP server.

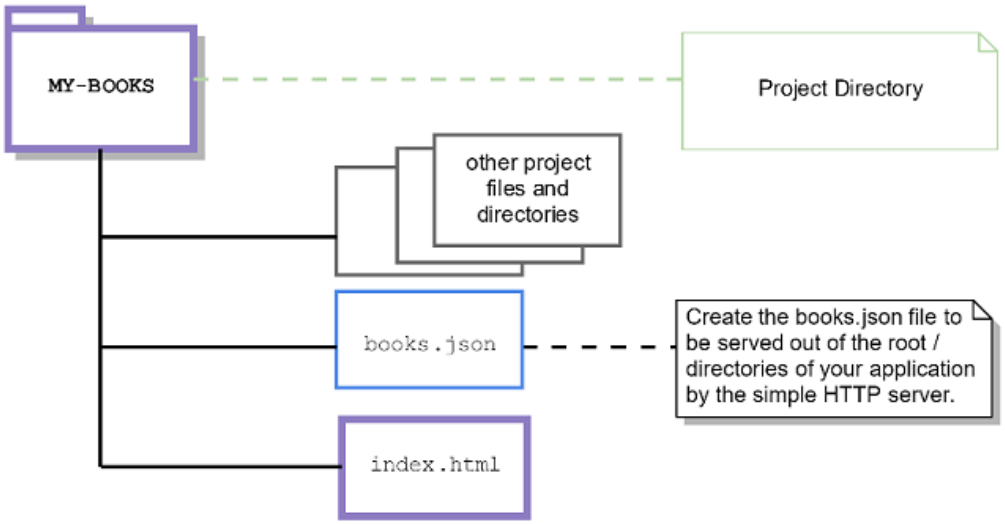

Figure 2.13 Add the books.json file to the root of the my-books project.

Copy and paste the code from listing 2.7 into the `/books.json` file.

Listing 2.7 JSON file containing initial my-books seed data – books.json

```
[ (1)
    {
        "Id" : 1,
        "title" : "War and Peace",
        "description" : "Really enjoyed this one.",
        "rating" : 3,
        "status": "good"
    },
    {
        "Id" : 2, "title" : "Oliver",
        "description" : "",
        "rating" : 2,
        "status" : "ok"
    },
    {
        "Id" : 3,
        "title" : "Charlie and the Chocolate Factory",
        "description" : "",
        "rating" : 5,
        "status" : "bad"
    }
]
```

(1) Our list of books stored as a simple JSON file on disk

If it's not still running, run the `au run --watch` command to initialize the Aurelia application. If you navigate to http://localhost:9000/books.json you should see the JSON results as shown in figure 2.14.

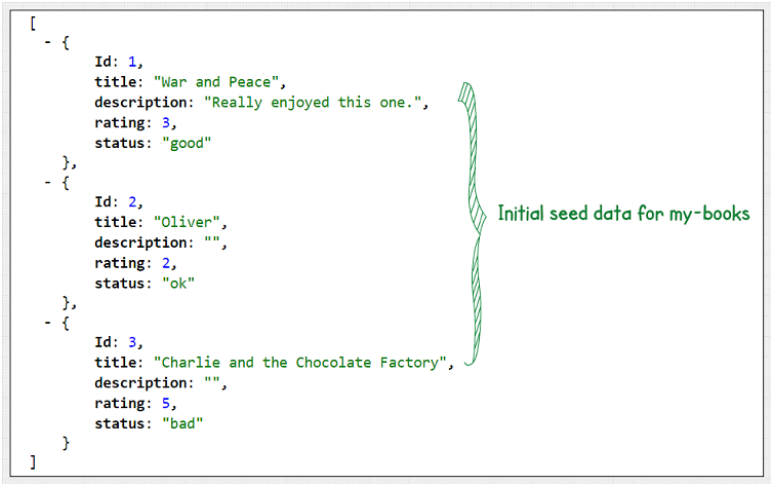

Figure 2.14 The contents of the books.json file should now be served by the HTTP server at http://localhost:9000/books.json.

The next step is to install the `aurelia-fetch-client`. This library provides the `HttpClient` module, which is an HTTP client based on the *fetch* standard. This is new way of talking AJAX (a native alternative to `XmlHttpRequest`), which you'll delve into in more detail in chapter 8. From a terminal/command prompt, run the `au install` command to install the library:

```
au install aurelia-fetch-client
```

You may be given the option of installing optional CSS files. In this case, the CSS files are only related to unit testing the `aurelia-fetch-client`, so select `[no - option 2]` to skip addition CSS file installation. You can think of this command as being a two-step process:

1. Download and install the package from the NPM repository.
2. Auto-configure the `aurelia.json` file to add the script to the *vendor* bundle.

Installing packages with the Aurelia CLI using the Aurelia Importer

There are two commands for installing packages with the Aurelia CLI. These packages leverage either NPM or Yarn (https://yarnpkg.com/en/) to download packages from the NPM repository and install them into your Aurelia project. The two commands are as follows:

`au install {package}` **(example: au install jQuery): Installs the package with NPM or Yarn and attempts to configure it by making the necessary changes to the aurelia.json file.**

`au import {package}` **(example: au import jQuery): Configures a package that has already been installed by NPM or yarn by making the necessary changes to the aurelia.json file.**

A part of the import step (step two of the `au-install` command), the new dependency needs to be added to the *vendor bundle.* When we generated the my-books project using the `au new` CLI command, the project build configuration was automatically created for us at `/aurelia_project/aurelia.json`. The `aurelia.json` file has several jobs related to how the Aurelia application is run. One of these is bundle configuration.

When the `aurelia-run` command runs, one of the things it does is transpile the ES6/7 code into ES5 so that it will run in older browsers and concatenate these converted files into clumped files called *bundles*. The maximum concurrent requests on browsers today ranges from 2 to 8. With an application composed of many small files you can imagine the effect that this has on page-load time. Reducing the number of HTTP requests to just these bundles helps to avoid this. The `au new` CLI command generates a configuration containing two bundles by default: The `app-bundle.js`, which contains your application code, and the `vendor-bundle.js`, which contains the libraries that you depend on.

As a result of running the `au install aurelia-fetch-client` command, the `/aurelia_project/aurelia.json` file for your project is updated, as shown in listing 2.8.

Listing 2.8 Modified project JSON file to include aurelia-fetch-client – aurelia.json

```
…
    {
    "name": "vendor-bundle.js",
    "prepend": [
     "node_modules/bluebird/js/browser/bluebird.core.js",
     "node_modules/requirejs/require.js"
    ],
    "dependencies": [
     "aurelia-binding",
     "aurelia-bootstrapper",
     "aurelia-dependency-injection",
     "aurelia-event-aggregator",
     "aurelia-fetch-client", (1)
     "aurelia-framework",
     "aurelia-history",
…
```

(1) Adds aurelia-fetch-client package as a dependency.

> **NOTE** Bluebird is third-party promise library with fantastic performance characteristics. According to tests done by the Aurelia core team, introducing the bluebird library into an Aurelia project improved application load times by approximately 25% in Google Chrome and approximately 98.5% in Edge, compared with the native promise implementations.

Now that you can retrieve data from the *books* API, create a new JavaScript file `/src/book-api.js`, as shown in listing 2.9. The `BookApi` client will fetch data from the back-end REST API and return it asynchronously via the `aurelia-fetch-client`.

Listing 2.9 Initial implementation of book API – book-api.js

```
import {HttpClient} from 'aurelia-fetch-client'; (1)
import {inject} from 'aurelia-framework';    (2)

@inject(HttpClient)    (3)
export class BookApi{

  constructor(http){    (3)
    this.http = http;   (3)
  }

  getBooks(){

    return this.http.fetch('books.json')     (4)
         .then(response => response.json()) (5)
         .then(books => {
           return books; (6)
         });

  }

}
```

1. Import the HttpClient class from the aurelia-fetch-client package.
2. Import the inject decorator from the aurelia-framework (used for dependency injection)
3. Inject the HttpClient class into the book-api class using dependency injection.
4. Make an HTTP GET call to the books.json endpoint using the fetch method.
5. The fetch call returns a promise. Un-wrap the promise with using the .json() method on the response object and extract the json result from the promise.
6. Take the JavaScript object representation of the object returned from the API call stored in the books object and return it to the caller

`BookApi` is a standard ES2015 class apart from the use of the `@inject` decorator. The first two lines import the required modules. Aurelia uses the concept of dependency injection to supply objects with the modules that the depend on at run-time. The `inject` module provides an Aurelia decorator, which indicates that the framework should supply an object.

After importing the requisite modules, the `@inject` decorator is used to supply a singleton instance of the `HttpClient` module as a parameter to the constructor.

> **NOTE** All injected instances in Aurelia are singleton by default within a given container where each component (such as a custom element or attribute) has its own container. It's possible to override this convention if needed.

That way, when the Aurelia framework constructs the `BookApi` class, this instance will be injected. Within the constructor, the injected `HttpClient` instance is saved into the `http` class property. The `getBooks` method returns a promise. This promise consists of performing an HTTP fetch out to the `books.json` endpoint, extracting the JSON from the `response` object and returning the resulting array of books.

> **THINGS TO COME** This is just one simple example of how the fetch client can be used. We'll dive into much more detail around the functionality provided by this package in chapter 8 when we look hook up the real back REST API for the my-books project.

Now that you have the functionality required for fetching the `books` array from the API endpoint, all you need to do is wire this up in the `app` view-model so that the stored list of books is retrieved and rendered to the user on startup. To include the logic to fetch a list of books from the REST API using the `BookApi` service class, modify the `/src/app.js` view-model as shown in listing 2.10.

In listing 2.10, we begin by importing the `BookApi` service and `inject` decorator. Then we inject an instance of the `BookApi` service into the `book-list` view-model. Finally, in the constructor call, the `getBooks` method on the injected `BookApi` service, the response of the promise returned from the `getBooks` method is unwrapped, and the resulting books are pushed into the `@bindable books` property.

Listing 2.10 Modified app view-model to retrieve books from API – app.js

```
import {bindable, inject} from 'aurelia-framework';(1)
import {BookApi} from 'book-api'; (2)

@inject(BookApi) (3)
export class App {

  constructor(bookApi){
    this.bookTitle = "";
    this.books = [];
    this.bookApi = bookApi;
  }

  addBook () {
    this.books.push({title : this.bookTitle});
    this.bookTitle = "";
  }

  bind(){ (4)
    this.bookApi.getBooks().then(savedBooks => (5)
    this.books = savedBooks); (6)
  }
}
```

(1) Import the inject decorator so that we can inject an instance of the BookApi class.
(2) Import the new Book-API class from the book-api module.
(3) Inject a singleton instance of the BookApi class into the constructor.
(4) Hook into the bind component life-cycle method.
(5) Wire up a method to be fired on completion of the getBooks request promise.
(6) Refresh books array with values returned from the HTTP API.

Calling an asynchronous method on a service class that returns a promise with the results of back-end web API call is a popular Aurelia framework pattern. This pattern can be used to perform any kind of interaction you might need with an HTTP endpoint. For example, in the context of my-books, you could also use this to delete books using the HTTP `delete` verb or modify books with the `put` verb. We'll delve into a variety of use cases for this service class, HTTP API combination in chapter 8, when we look in depth at using HTTP in Aurelia.

If you're still following along, kill the `au run --watch` process and re-launch it to restart your HTTP server. Your browser should then pick up the new dependency on `the aurelia-fetch-client` and refresh with a new list of books loaded from your API, as shown in figure 2.15.

Figure 2.15 The book list rendered inside the book-list component is now made up of the books returned from the books.json HTTP endpoint.

TROUBLESHOOTING STEPS

If you encounter any errors at this point it's worth checking the following:

- Ensure that the `aurelia.json` contains the `aurelia-fetch-client` dependency.
- Ensure that you've imported the `book-api` service into your `book-list` view-model and that the import path is correct (you'll receive an HTTP 404 error if the path to this import is incorrect.

The book-list component is now dynamic, loaded from an HTTP API rather than being hard-coded in the view-model. However, looking at the results in figure 2.15, the layout still doesn't feel quite right. The my-books application is now only about adding and listing books, but you'll want to add more pages in the future to expand the functionality of the application. In the next step, we'll shift the responsibilities for adding and listing books into separate application routes using the Aurelia router. This gives us a more flexible application layout that we can expand on throughout the book.

2.6 Maintaining Page State with the Aurelia Router

Have you ever had the experience where you follow through all the steps in a web-application, getting close to finishing the form, and then need to click the back button to double-check something on a previous page? All too often, when you do this the entire workflow you were going through loses its state and resets back to the beginning. This tends to frustrate users

immensely. To remedy this, a modern single-page application framework needs a powerful routing system that is easy to implement.

The next thing that my-books needs is a way of maintaining state as we transition through different logical pages of the application. We'll begin by implementing the Aurelia router. When it's in place, we'll set up a home view that the user initially sees when they load the page.

2.6.1 Configuring the Router

The Aurelia router implementation consists of two parts:

- The `<router-view>` custom element: The routable area consists of anything inside this element. This should be added to the component you want to route inside.
- The view-model of this component needs to include a `configureRouter` function which defines the routes that your application supports.

The layout of the application needs to be changed slightly to support the new routing system. To do this, you'll extract the book retrieval functionality into a new `books` component, and strip the `app` component down to a basic shell. Replace the contents of the `/src/app.js` file as shown in listing 2.11 to include the router.

Listing 2.11 app view-model modified to include the route configuration – app.js

```
export class App {
  configureRouter(config, router) { (1)
  this.router = router; (2)
  config.title = 'my-books'; (3)
  config.map([ (4)
   { route: ['', 'home'], name: 'home', moduleId: 'index' }, (5)
   { route: 'books', name: 'books', moduleId: 'books' },     (6)
  ]);
  }
}
```

(1) Define the route configuration function
(2) The router on the view-model
(3) Set the view title
(4) Add the routes to the router
(5) Define the home route
(6) Define the books route

The `configureRouter` function defines which routes the application supports. First, the router of the current view-model is set to the injected router instance. Following this, the title of the application is configured. The map function is then provided with the current list of available routes. We are configuring only some basics in this example. Each route listed defines the following:

- `route`: The actual URLs that this route listens for.
- `name`: The name of the route is defined so that it can be referenced when setting up links in views.

- `moduleid`: The name of the component that this route should initialize.

Now that you've got the initial set of application routes configured, the next step is to make use of these new routes in the application layout. You'll modify the application layout to change the `app` component into a shell that hosts several views. We do this by including a reference to the `<router-view>` custom element. This element serves as a slot in the page where the Aurelia router can swap in content at run-time based on the active route. This is what gives the appearance that we're navigating to a different page, as we would in a traditional server-based web application, without the overhead of needing to perform a full page-refresh.

Modify the `/src/app.html` view file so that it includes only the `router-view` as shown in code listing 2.12.

Listing 2.12 app view simplified to a router-view shell – app.html

```
<template>
 <router-view></router-view> (1)
</template>
```

(1) Add the router-view shell to the app view

Include the `<router-view>` custom element. This is the shell that hosts the views that correspond to the routes defined in your array of routes. The Shell is now configured, but the new home page that you configured in the router setup isn't available yet. Create a new `/src/index.js` view-model file, as shown in listing 2.13. This is relatively empty for now.

Listing 2.13 Initial implementation of index view-model – index.js

```
export class Index{   (1)
}
```

(1) Empty index.js view-model class.

The `index.js` file is the view-model for the home page of our application. At this stage, it's bare bones as we don't need any functionality here at this point. By now you should be starting to become familiar with the steps for creating custom elements in Aurelia. Now that we've got the view-model created, add the corresponding view as shown in listing 2.14 to complete the `/src/index.html`. This view will serve as the my-books home page.

Listing 2.14 Initial implementation of index view – index.html

```
<template>
  <h1>my-books</h1>

     <p>
                my-books allows you to keep track of the books
                you've read by adding and rating them as you read.
        </p>
```

```
  <a route-href="route: books;">books</a> (1)
  </template>
```

(1) Reference to stored route

The `route-href` expression creates an HREF to the route stored against the `books` key in the route array using Aurelia custom attribute called `route-href` referencing the books named-route that you defined in the router configuration.

Next, create a view, view-model pair to serve as the `books` parent component, which now provides the functionality that was previously provided by the `app` component:

Create the `/src/books.js` view-model using the code in listing 2.15. This becomes the new host of the book-management logical that previously sat in the `app.js` view-model as a part of encapsulating all the book management functionality into a new page.

Listing 2.15 Initial implementation of books view-model – books.js

```
import {bindable, inject} from 'aurelia-framework';
import {BookApi} from 'book-api';

@inject(BookApi)
export class Books { (1)

  constructor(bookApi){
    this.bookTitle = "";
    this.books = [];
    this.bookApi = bookApi;
  }

  addBook () {
    this.books.push({title : this.bookTitle});
    this.bookTitle = "";
  }

  bind(){
    this.bookApi.getBooks().then(savedBooks => this.books = savedBooks);
  }
}}
```

(1) Refactor book management functionality from app view-model into books view-model

Create the corresponding `/src/books.html` view. This contains the functionality moved across from the `app.html` page. All the book-management component references have been moved across to this new encapsulated book management page.

Listing 2.16 Initial implementation of books view – books.html

```
<template>
    <require from="./book-list"></require> (1)
    <h1>Books</h1>
    <form submit.trigger="addBook()">
        <label for="book-title"></label>
```

```
        <input value.bind="bookTitle"
            id="book-title"
            type="text"
            placeholder="book title...">

        <input type="submit" value="add">
    </form>
    <hr/>
    <book-list books.bind="books"></book-list> ❷
</template>
```

❶ **Import the book-list custom element**
❷ **Use the book-list custom element**

Next, include a reference to the `book-list` component that we created in the previous section, and move the custom element reference into the books page which now hosts all book management related functionality.

You've now completed all the layout refactoring required to move the book management functionality into its own page. If you're following along, the application should now look like the image in figure 2.16 when it's initially loaded in the browser.

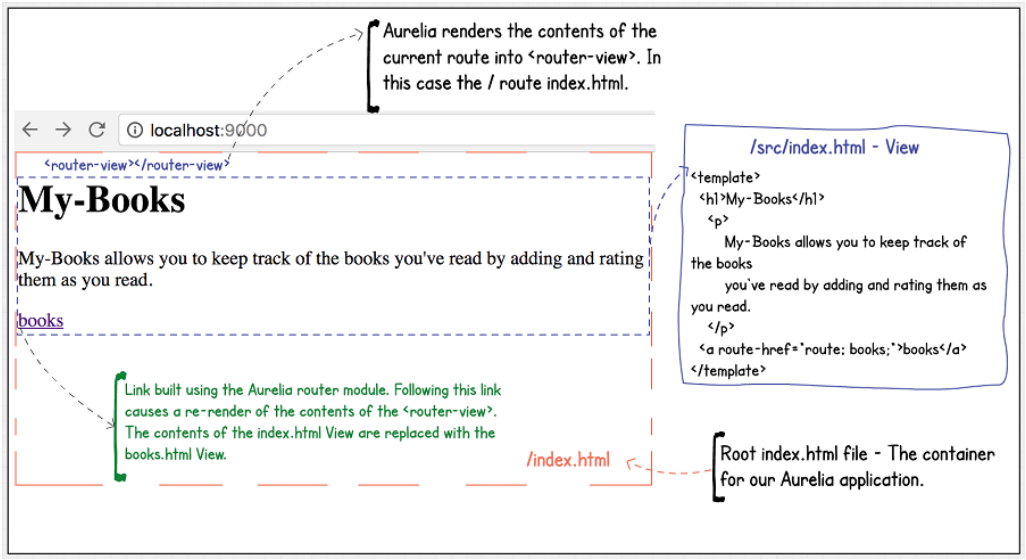

Figure 2.16 my-books home route can be reached at http://localhost:9000 or http://localhost:9000/#/home. When you navigate to this route, the contents of the <router-view> custom element are re-rendered with the contents of the index.html view template.

The my-books application layout has now been refactored to make use of the Aurelia router. The main index page serves as a starting point for the user and gives us an area to add more

functionality as the application progresses. The book-management functionality has been refactored into a separate Books page which is available via a link on the index page.

You can click the new *books* link to navigate the browser to the `books` route at http://localhost/#/books.

Troubleshooting

If you receive an error after creating a new route like our *books* route from listing 2.14, ensure that your routes are defined correctly in the `configureRoute` function that you've defined in the view-model that backs the view containing the `<router-view></router-view>`

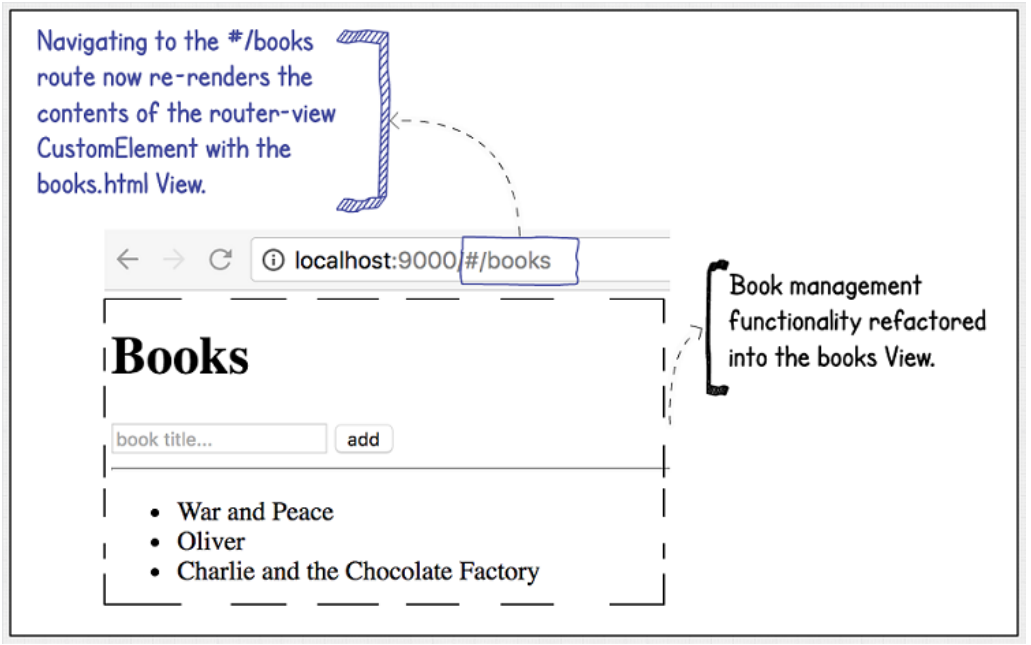

Figure 2.17 The book-management functionality has been encapsulated into a new 'Books' page. This page includes a 'book.js' view-model that is indexed under the 'books' route in our array of routes configured in the app.js file. When the user navigates to /books, the content of the <router-view> in the app.html page is re-rendered with the contents of the 'Books' view.

With a few additional lines of code, my-books is starting to behave like a fully fledged single-page application. Users can use the back and forward buttons to transition between the two logical states of the application. This just scratches the surface of what the Aurelia router can do. I'll cover it in a lot more depth in chapter 10.

2.7 Moving Forward

So far, we've structured the my-books project in the simplest way possible, with all views and view-models stored at the `/src` directory. Although this was fine to begin with, you can imagine how out of hand things could become as the application grows. Before we launch into chapter 3, let's reorganize the project structure to be more in line with Aurelia convention. The Aurelia project structure under `/src` currently looks like figure 2.18:

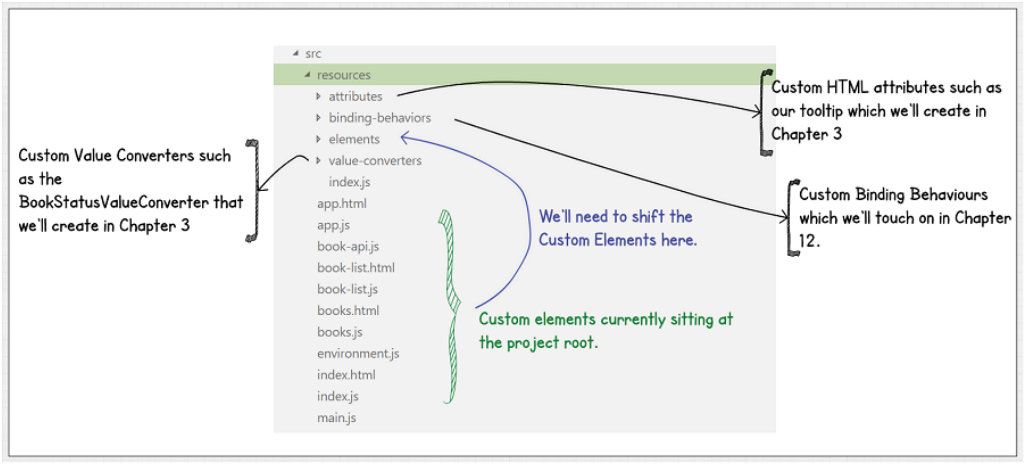

Figure 2.18 Current my-books directory structure with all views, view-models and services sitting under the src directory.

Eager to get to Chapter 3?

If you want to skip this re-organization exercise and jump directly into chapter 3 you can check out the Aurelia in Action GitHub repository using the following command: git clone https://github.com/freshcutdevelopment/Aurelia-in-Action.git. From there you can get the completed chapter 2 my-books project under \Chapter-2-Complete.

There are several directories under the `resources` folder. Each of these directories should be used for the relevant kind of view resource (an external resource such as a custom element that you `require` into a view). We'll go into each of these view resource types in detail in chapter 3, but for now we only need to concern ourselves with the custom element resources that we've created so far. Let's start by moving the custom elements created in this chapter under the `resources/elements` directory. Next, create a new directory `src/services` and move the `book-api.js` service into it. Once you're finished the directory structure should look like figure 2.19:

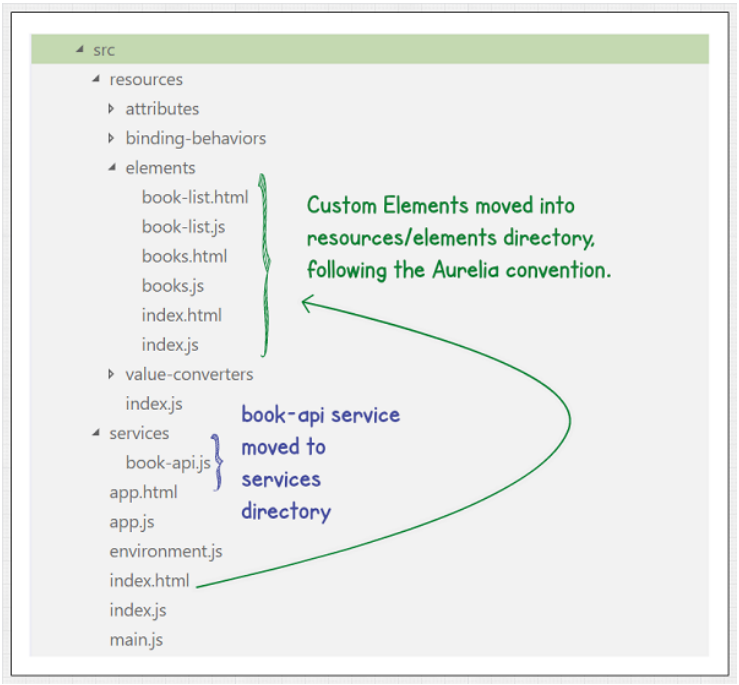

Figure 2.19 Reorganize the my-books src directory structure by moving all custom elements under the /src/resources/elements directory and moving the book-api.js service under a new directory /src/services.

Now that the files are where they should be we need to modify some of the paths in our `app.js` and `book-list.js` view-models to use the new locations. Modify the `/src/app.js` file as shown in listing 2.17 to correct the books module path.

Listing 2.17 app view-model modified to fit new directory structure – app.js

```
export class App {
  configureRouter(config, router) {
  this.router = router;
  config.title = 'My-Books';
  config.map([
   { route: ['', 'home'], name: 'home', moduleId: 'index' },
   {
     route: 'books',
     name: 'books',
     moduleId: './resources/elements/books' (1)
   },
  ]);
  }
}
```

(1) **Modified moduleId to point to new location of books module.**

Finally, we need to modify the `/src/resources/elements/books.js` file to import the `book-api` service from the services directory as shown in listing 2.18.

Listing 2.18 book-list view-model file modified to import book-api service from the services directory – books.js

```
import {bindable} from 'aurelia-framework';
import {BookApi} from '../../services/book-api'; (1)
import {inject} from 'aurelia-framework';

…
```

(1) book-api service changed to import from the services directory

2.8 my-books project status

In this chapter, we've created the initial implementation of the my-books virtual bookshelf. We've used routing to split the application into two pages, with the Aurelia router loading he correct view for a given URL. We've used Aurelia's binding commands and template repeaters to give users the ability to add books to a list, and view the updated shelf contents as books are added. The final state of the project at the end of this chapter is depicted in figure 5.20.

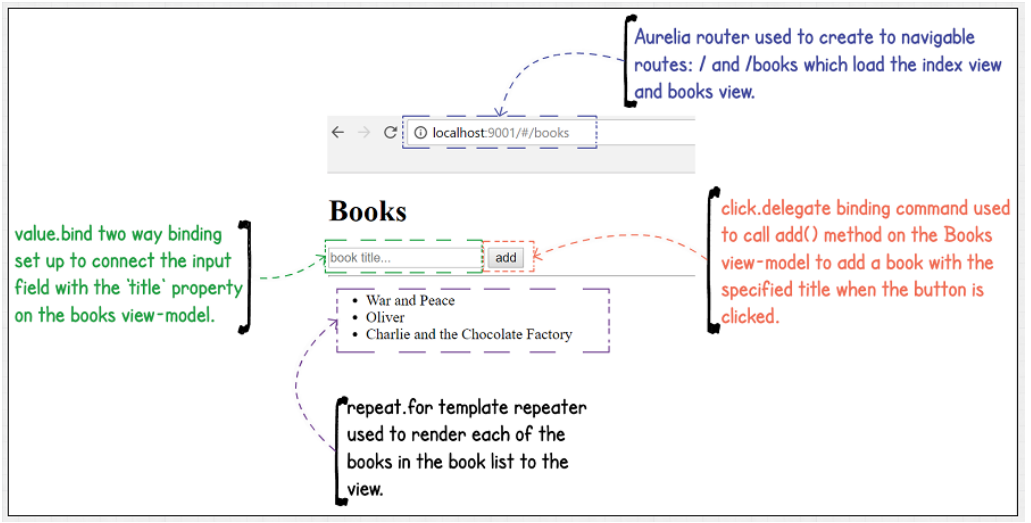

Figure 2.20 my-books book management page state at the end of chapter 2.

You can download the completed project by from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

2.9 *Summary*

In summary, we've covered:

- There are three main options for creating Aurelia applications: The *quick-start* download, the *skeleton projects,* and the Aurelia CLI.
- The Aurelia CLI is a Node.js application that allows you to easily create, run and deploy Aurelia projects.
- We introduced a the my-books virtual book-shelf sample project that we'll follow throughout the book.
- Aurelia applications are built up of components, such as custom elements.
- Custom element components generally consist of a view, view-model pair where the view is an HTML template and the view-model is an ESNext (or Typescript) class. Except in the case of HTML only custom elements.
- Aurelia uses binding to propagate changes in data from the view to the view-model and Visa-Versa.
- Binding is also used to transmit events raised on the view to corresponding view-model methods.
- Operations such as retrieving data from REST APIs are typically implemented in Aurelia using standard ES2015+ classes. This keeps Aurelia's concept count low, as the bulk of an application is not Aurelia specific.
- The `aurelia-fetch-client` plugin allows easy communication with back-end services over HTTP using standard verbs such as GET, or POST.
- The Aurelia router allows you to map Aurelia views to corresponding URLs. As the user navigates the application the window location is updated with the current views URL. This means that standard browser features such as the back button to work as expected. Each time the route changes the content of the `<router-view>` component is re-rendered with the contents of the current view
- The Aurelia CLI creates a base project structure that by convention gives us a reasonable place to put the various files that make up our application.

Like a tourist visiting Rome for the first time you've been on a whirlwind tour of the Aurelia framework. You've seen all the major attractions but have a feeling that you're only scratching the surface of what it has to-offer. In the next chapter, we'll expand on the book-management functionality that we added in chapter 2 enhancing the book-management form by adding constraints around when books can be added, and the ability to remove books from your library. In doing so we'll dive into some of the more powerful features of Aurelia's templating and binding system.

3

View resources, custom elements, and custom attributes

This chapter covers

- **View resources**
- **Custom elements**
- **Custom attributes**

The core of the Lego toy ecosystem is the 8*8*10mm Automatic Binding Brick, originally called the Murnsten, which almost all other LEGO construction toys are derived from. These standard Lego bricks can be combined to produce a multitude of different creations by millions around the globe (up to 915,103,765 different combinations according to Wikipedia: https://en.wikipedia.org/wiki/Lego#Design). What makes this toy different—and makes Lego one of the most successful toy companies in the world today—is that kids aren't limited to what the toy designers have imagined for them, but rather by only their imagination. The same bricks can be used to build a car, a house, or a castle, simply by combining them in diverse ways. In the world of programming, we'd call the Murnsten a *primitive*, a core building block that can be used to create a more complex system.

Most of the more successful technologies historically have been built by defining these core primitives, and building on top of them. A famous example of this is HTML. HTML provides us with a set of simple components like `<h1>`, `<body>` or `<span>`, which combined with CSS and JavaScript can be used in many ways to produce any site on the web.

3.1 Understanding Aurelia's templating primitives

Aurelia follows the same tactic. In this chapter, we'll look at some of the core primitives of the Aurelia templating system, how the Aurelia framework builds on these primitives to provide higher level features, and how we can use these features to build Aurelia components. The primitives explored in this chapter include the CSS resource, custom elements, and custom attributes, all of which inherit from the *view source* primitive.

3.1.1 Aurelia's templating primitives

One of the core primitives of Aurelia's templating system is `ViewResource`. View resources are resources that you `<require>` into an Aurelia view to extend it with additional functionality.

Figure 3.1 provides an overview of the anatomy of the core primitives that make up Aurelia's templating engine. This serves as a mental model for how each of the primitives in Aurelia's templating and binding system fit together. Don't be concerned if it looks complicated at first glance. There is method to my madness. As we explore the various easy-to-use components of the Aurelia framework, I'll use this architecture to show you where each component fits in the bigger picture.

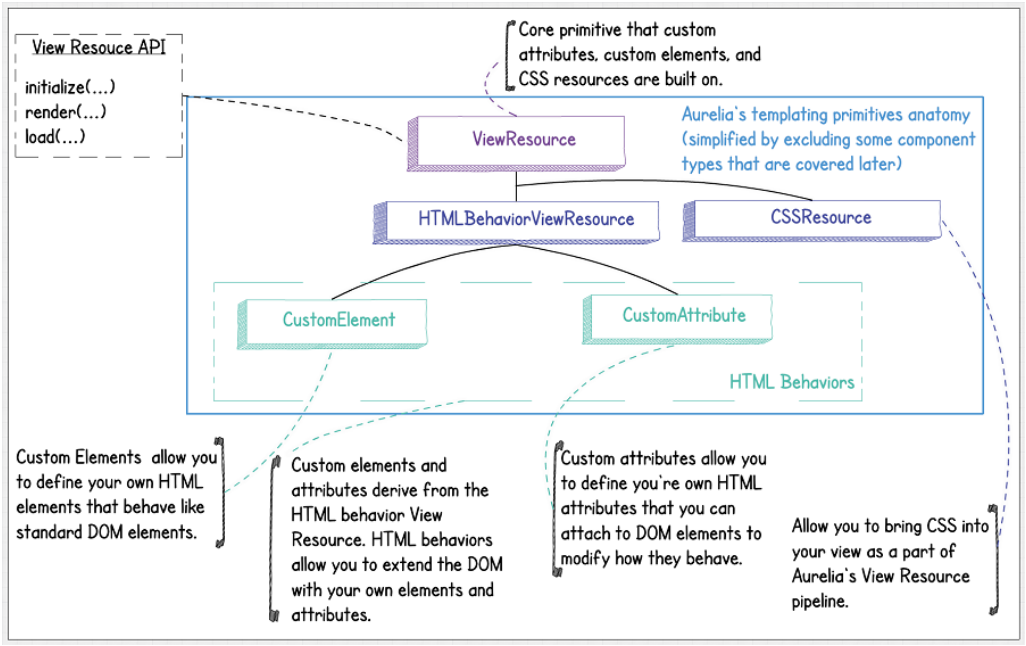

Figure 3.1 Pages in Aurelia are built by composing various kinds of resources—external items that are loaded into the view to extend it with additional functionality. These resources all derive from an Aurelia framework primitive called a view resource.

We'll start by adding some much-needed style to my-books, and in doing so, you'll get an overview of a subset of core view resource types in the Aurelia framework: the `HTMLBehaviourResource` and `CSSResource`. Under the hood, all resources implement the basic view resource API with methods to tell the framework how they should be initialized, rendered into the DOM, and dynamically reloaded. The `CSSResource` allows you to `<require>` a CSS file into your view. `CustomElement` and `CustomAttribute` view resources both inherit from a primitive called the `HtmlBehaviourResource`. This is not the complete picture. Other view resources such as values converters were left out for simplicity.

3.2 CSS resources

Let's begin our tour of the Aurelia templating primitive's system by looking at the CSS resource. In this section, we'll use the CSS resource to add a nice coat of paint and polish to our virtual bookshelf.

3.2.1 Setting up the project dependencies

Before we begin adding custom styles using Aurelia CSS resources there is some setup work we need to do. The style changes we're going to make in this section depend on two external dependencies:

- **Font awesome:** An open source collection of scalable vector icons. We'll use this to add icons to buttons and so on. You can find out more detail about Fontawesome and browse the catalogue on the project website. http://fontawesome.io/.
- **Bootstrap 4 Alpha:** Because we like staying on the bleeding edge we'll also be using the latest version of the Bootstrap HTML, CSS and JavaScript framework. We'll style our application by using some of the layout and style classes provided by the Bootstrap CSS library, and use Bootstrap jQuery plugins to enhance parts of our user interface without needing to write everything from scratch. Styling an application with Bootstrap is not within the scope of this book, but you can find out more information on the project website. https://v4-alpha.getbootstrap.com/

For simplicity, we'll include these dependencies by adding references to the Fontawesome and Bootstrap CDNs (Content Delivery Networks-hosted version of these files) into the `index.html` file. We'll also include the required Bootstrap dependencies: tether (used under the hood by the Bootstrap tooltip jQuery plugin), and jQuery. This ensures that the dependences are loaded before the Aurelia application is initialized. (If you want to see how to add these dependences using Aurelia's require custom element instead, see appendix A.) Modify the `./src/index.html` file as shown in listing 3.1 to include the new dependencies.

Listing 3.1 Modify index.html file to include Bootstrap and Fontawesome – index.html

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <meta charset="utf-8">
    <title>Aurelia</title>
    <link href="https://maxcdn.bootstrapcdn.com
                    /font-awesome/4.7.0/css/font-awesome.min.css"
          rel="stylesheet" > (1)
    <link rel="stylesheet"
          href="https://maxcdn.bootstrapcdn.com
                /bootstrap/4.0.0-alpha.6/css/bootstrap.min.css" > (2)

  </head>

  <body aurelia-app="main">
    <script src="https://code.jquery.com/jquery-3.1.1.slim.min.js"> (3)
    </script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs
                     /tether/1.4.0/js/tether.min.js"> (4)
    </script>
    <script
        src="https://maxcdn.bootstrapcdn.com/bootstrap
                  /4.0.0-alpha.6/js/bootstrap.min.js"> (5)
    </script>
    <script src="scripts/vendor-bundle.js"
            data-main="aurelia-bootstrapper">
    </script>
  </body>
</html>
```

(1) **Include Font Awesome CSS via CDN**
(2) **Include Bootstrap 4 CSS via CDN**
(3) **Include jQuery JavaScript (required by Bootstrap)**
(4) **Include tether JavaScript (required by Bootstrap)**
(4) **Include Bootstrap 4 JavaScript**

With the dependencies set up we're ready to add a custom stylesheet to the `app` component.

3.2.2 Adding the CSS resources

First, download the my-books CSS file from Github https://github.com/freshcutdevelopment/Aurelia-in-Action/blob/master/Chapter-3-Complete/my-books/src/styles.css. Copy this file to under the `src` directory of the my-books project. To begin using the `./src/styles.css` file, we'll need to look at an Aurelia view resource that makes it easy to add styles to components as a part of the Aurelia's view resource pipeline.

An Aurelia `CSSResource` is a special kind of view resource that allows you to load stylesheets related to a component. There are two steps we need to take to add styles to our application:

1. Create the stylesheet
2. Use the `<require>` custom element to load the stylesheet into our component.

Because these styles apply to our entire application, we're going to include them in the `app` component. Modify your `app.html` file as shown in listing 3.2 to require the style resources for our custom styles.

Listing 3.2 Modify app view to include the stylesheet – app.html

```
<template>
    <require from="styles.css"></require> ❶
    <router-view></router-view>
</template>
```

❶ Import the custom CSS

By including a CSS resource in our application, we tell Aurelia to inject a new `<style>` element in the location of our `<require>` custom element. Then when the `app.html` file is loaded, Aurelia will go through the following steps to load the child view resources:

1. Load the style.css CSS resource
2. Load the `<router-view></router-view>` custom element, and based on the current route, load the appropriate component. For example, if we visit the URL http://localhost:9000#/books the `<books></books>` custom element is loaded.

Figure 3.2 depicts the steps that Aurelia goes through to load each of the view resources on the `app.html` page, and recursively traversing through the child components.

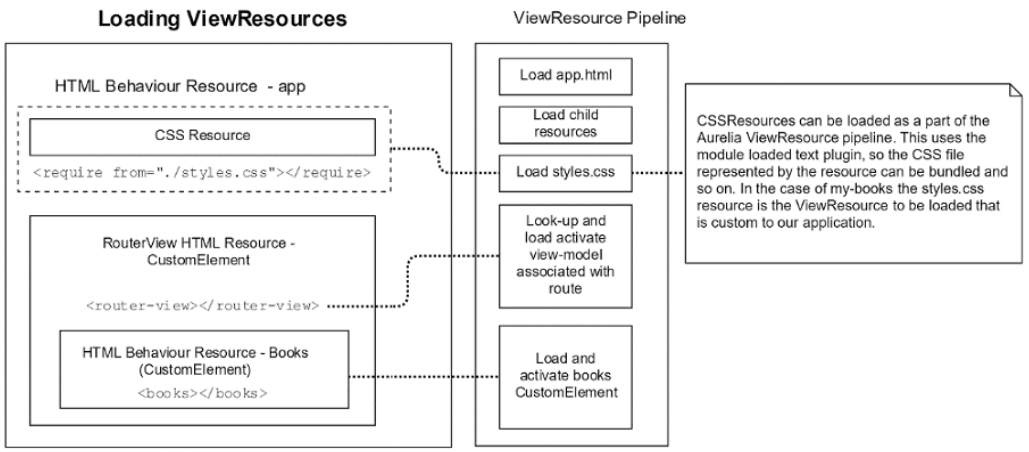

Figure 3.2 Aurelia views are loaded by recursively traversing through the DOM tree and loading all child resources found in each nested view. This is a simplified view because there are also child components underneath the <books></books> element.

This process of recursively loading view resource is the view resource pipeline. Examples of view resources loaded in this way include custom elements, and CSS Resources. Loading this

resource causes a new `<style>` element to be injected into the header of the `<head>` element in the DOM containing the contents of the `styles.css` file. You can inspect this behavior using the browser dev tools (F12 in Chrome) as shown in figure 3.3.

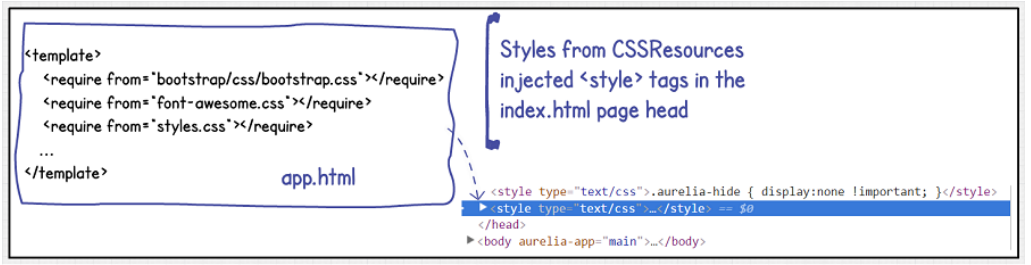

Figure 3.3 CSS Resource processed and a new style tag is injected into the DOM containing the contents of the file referenced by the CSS Resource.

The CSS file contents are loaded by the module loader plugin that has been wired up in your Aurelia application. In our case, for example, the contents of the CSS file are loaded by RequireJS. This also means that they can be bundled as a part of the `app-bundle.js` file as an alternative to including them inline as we've done here.

Including the styles in this way works well if we don't mind our styles being applied to our entire Aurelia application (default behavior for referenced CSS files). But, what if you want to create a component (such as an email sign-up widget) that would be styled in its own way, with the styles outside of the component not leaking into it, and vice versa? In Aurelia, we can achieve this using an exciting feature called *scoped CSS resources*. Scoped CSS resources inject your CSS into the *Shadow DOM* (a new feature available as a part of the Web Components specification). CSS injected into the Shadow DOM is only applied to the contents of the *shadow root* element (the root node of a Shadow DOM tree) for the component. We'll delve into *Scoped CSS resources* in-depth later in this chapter when we look at how to use web component features as a part of our Aurelia development workflow. Now that we've included these CSS resources let's modify each of our views to make use of these styles.

3.2.3 Styling the my-books views

In this section, we'll modify each of the views to use the new styles. The following views need to be updated to take advantage of the new them.

- `app.html`
- `book-list.html`
- `books.html`
- `index.html`

If you'll bear with me and replace each of these views with the contents of listing 3.3 to listing 3.6 we should end with an application that behaves the same but follows the my-book theme.

Listing 3.3 Modify app view file to include new styles and nav bar – app.html

```
<template>
    <require from="bootstrap/css/bootstrap.css"></require>
     <div class="container">
        <div class="header clearfix">
        <h3 class="text-muted">
          <span class="brand-highlight">my </span>
          books
      </h3>
      </div>
      <router-view></router-view>
      <footer class="footer">
        <p>&copy; Aurelia Demo 2017</p>
      </footer>

    </div>

</template>
```

Listing 3.4 Modify book-list view to use the new styles – book-list.html

```
<template>
    <ul class="books list-group list-group-flush">
      <li class="list-group-item" repeat.for="book of books">
        <div class="col-10">${book.title}</div>
      </li>
    </ul>
</template>
```

Listing 3.5 books.html view modified to use the new styles – books.html

```
<template>
 <require from="./book-list"></require>

 <h1 class='page-heading'>books</h1>
       <div class="card">
        <div class="card-block">
          <form class="form-inline" submit.trigger="addBook()">

            <label for="book-title"></label>

            <input class="form-control" value.bind="bookTitle"
                id="book-title"
                type="text"
                placeholder="book title...">

              <input class="btn btn-success tap-right"
                       type="submit"
                       value="add">
          </form>
        </div>
    </div>
```

```
 <hr/>
 <book-list books.bind="books"></book-list>
</template>
```

Listing 3.6 Modify index view to use the new styles – index.html

```
<template>
      <div class="jumbotron">
        <h1 class="display-3">
          <span class="brand-highlight">my</span>
          Books
        </h1>
        <p class="lead">
          My-Books allows you to keep track of
          the books you've read by adding and rating them as you read.
        </p>
        <a route-href="route: books;">books</a>
      </div>

      <div class="row">
        <div class="col-lg-12">
            <p></p>
        </div>
      </div>
</template>
```

Now that we've styled my-books and it's starting to look more like a real application, it's time to start enriching the `books` and `list-books` components that we created in chapter 2. We'll supplement these elements with a suite of additional elements to allow us to encapsulate the new behaviour as we introduce it into our application. We'll also look at a new kind of view resource that we've not come across yet called the custom attribute, and use it to add tooltips across the buttons in our application.

3.3 Custom elements

HTML behavior resources include the custom element (mentioned in chapter 2 when you defined the `books` and `list-books` components), and the custom attribute. HTML behavior resources allow you to extend your HTML by providing your own HTML elements and attributes.

> **NOTE** Custom element and custom attribute classes are named using the standard JavaScript class naming convention of InitCase. InitCasing is where the first letter in each word of the class name is capitalized, for example: `BookList`. The HTML view, and JavaScript files on the other hand are named using kebab-case for example `book-list.html`, where the name is made up of lower-cased words delimited by a hyphen. The reason for this is HTML is not case sensitive.

3.3.1 Custom element explained

Custom elements—like the name indicates—allow you to define your own HTML elements and use them within an Aurelia view. These are built by creating an HTML view, view-model pair. Custom elements are the default type of HTML behavior resource. This means that although you can optionally provide the postfix `CustomElement` (for example we could have named the `BookList` custom element `BookListCustomElement` and the result would have been identical), if we leave this postfix out, the Aurelia framework assumes that this is a custom element by default and hence will treat it like a custom element in terms of how it is registered and rendered into the view.

3.3.2 My-books custom elements

So far, we've encountered two major custom elements while developing the my-books application: `<books>` and `<book-list>`, as shown in listing 3.7. This listing provides a convenient high-level overview of the major building blocks of the my-books page:

- `book` custom element (orchestrator: loads books from the API, passes them to the `book-list` component, and allows books to be added to the array)
- `book-list` custom element (renders the list of books to the screen)

Listing 3.7 books custom element view – books.html

```
<template> (1)
  <require from="./book-list"></require>     (2)
  …
  <book-list books.bind="books"></book-list> (3)
</template>
```

(1) Book custom element view template
(2) Import the book-list custom element
(3) Use the book-list custom element

To start, we use the `<require>` custom element to load the `book-list` custom element resource into the view. In doing this, we instruct the Aurelia framework to search the view for any elements called `<book-list>`. When it finds them, it follows the default convention of looking for a corresponding JavaScript class called `BookList` or `BookListCustomElement` (by taking the name of the module and InitCasing it). Views in Aurelia can be built up by iteratively creating custom elements using the following pattern.

CUSTOM ELEMENT CREATION STEPS:

1. Create a new JS file (for example `book-list.js`) that exports the view-model class (for example `BookList`) following the InitCase naming convention
2. Create the corresponding HTML view file—for example `book-list.html` following the *kebab-case* naming convention.
3. Import the custom element into the page using the `<require>` custom element.

4. Use the custom element in the page.

> **NOTE** The from attribute on the `<require>` element can either take a relative path from the current view as shown in listing 3.7, or it can take an a relative path from the root of the application (for example `<require from='./resources/elements/book-list'>`).

3.3.3 HTML-only custom elements

In the example custom elements we've seen so far in my-books, we've always created both a JavaScript class and corresponding HTML view file. This is useful when you have behavior that needs to be implemented in the class. For example, in the `Books` class, we needed to implement the behavior that was fired because of the `add` button being clicked as shown in listing 3.8.

Listing 3.8 View-model for the books component – books.js

```
import {bindable, inject, computedFrom} from 'aurelia-framework';
import {BookApi} from '../../services/book-api';

@inject(BookApi)
export class Books{ (1)
  ..
   addBook () {
       this.books.push({title : this.bookTitle});
       this.bookTitle = "";
   }
  ..
}
```

(1) view-model required to handle the Add button click behavior

But what about the case where we just want to extract a fragment of the view markup that doesn't necessarily have any corresponding non-view behavior? In this case, we can use an HTML-only custom element. HTML-only custom elements allow you to extract a section of the markup—also known as a *DOM fragment*—from the page without the need to implement a corresponding view-model class.

To demonstrate this functionality let's refactor the `books.html` view, moving the heading in its own HTML-only custom element. The following steps are required to refactor a part of a parent view into an HTML-only custom element:

HTML-ONLY CUSTOM ELEMENT CREATION STEPS:

1. Create a new HTML file to hold the extracted markup.
2. Move the relevant markup from the parent view into the newly created HTML file.
3. Require the newly created HTML file into the parent view.
4. Reference the custom element tag in the view.
5. Set up any bindings required to pass data from the parent view to the child view.

6. Use the new element.

STEP 1 – CREATING A HEADING.HTML FILE

Start by creating a new file called `heading.html` under the path `src/resources/elements/heading.html`. This HTML file is the new home of your refactored markup. When completing this step, start by including an empty set of `<template>` tags in the new file immediately on creation. Otherwise it's easy to forget to add them later and wonder why they view isn't being rendered correctly into the DOM. The newly created file should look like listing 3.9.

Listing 3.9 Heading view empty apart from <template> tags – heading.html

```
<template>
</template>
```

STEP 2 – MOVING THE MARKUP

Move the heading from the `books.html` file into the new `heading.html` file inside of the template tags as shown in listing 3.10.

Listing 3.10 Modify heading view to include hard coded heading – heading.html

```
<template> (1)
   <h1 class='page-heading'>books</h1> (2)
</template>
```

(1) <template> tags indicate that this should be handled as an Aurelia view.
(2) Extracted heading moved from books.html view.

STEP 3 – IMPORTING THE VIEW

To indicate to the Aurelia framework that this new child-view resource should be loaded as a part of the Aurelia templating pipeline, import the new view into the parent view (`books.html`) by adding a new require statement to the top section of the view as shown in listing 3.11.

Listing 3.11 Import HTML-only heading.html into books view – books.html

```
<template>
    <require from="./book-list"></require>
    <require from="./heading.html"></require> (1)
…
</template>
```

(1) New require statement to import the heading.html HTML-only view.

One important thing to note here is that we've included the `.html` extension in the from attribute on the `<require>` element. When using the `<require>` element on HTML-only custom elements, you need to include the file extension of the view to indicate that it is an

HTML-only custom elements and Aurelia should not follow the default convention of searching for a corresponding JavaScript file.

> **NOTE** When using `<require>` element on HTML-only custom elements you need to include the file extension of the view to indicate that it is an HTML-only custom element and Aurelia should not follow the default convention of searching for a corresponding JavaScript file.

STEP 4 – REFERENCING THE <HEADING> ELEMENT

Now that we've created new `<heading>` custom element, we need to include a reference to it. Replace the hard-coded heading text with the `<heading>` custom element tag as shown in listing 3.12.

Listing 3.12 Modify books view to reference new heading.html element – books.html

```
<template>
…
<require from="./heading.html"></require>
<heading></heading> (1)
…
</template>
```

(1) Reference the newly created <heading> element.

If you're still following along, you can view the results in the browser by running `the au run --watch` command in the terminal at the command-prompt and navigating to http://localhost:9000/#/books

The heading should display the same as it did before we refactored. Checking the Chrome Developer tools (F12), you should see that the `<h1>` now sits under the custom element tag in the DOM—highlighted in figure 3.4.

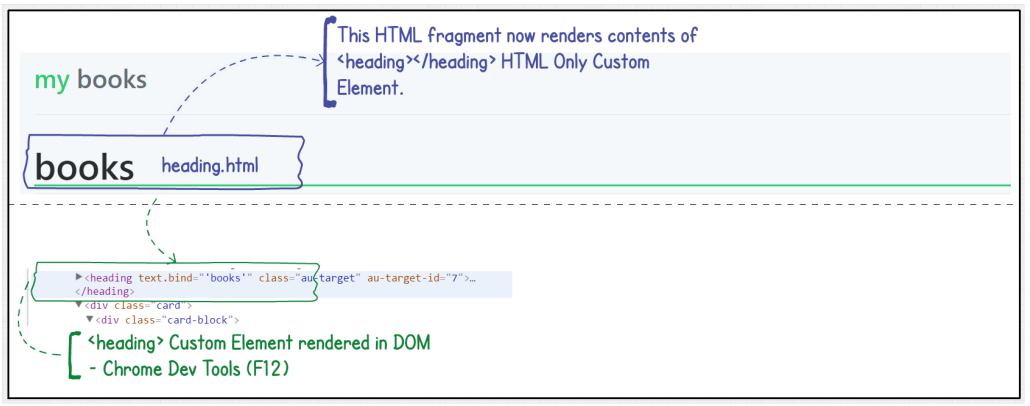

Figure 3.4 The page header should render just as it did previously, even though it is now being loaded from an

external view resource rather than being an in-line element within the view. Chrome developer tools view show that the <h1> tag now lives under the custom element.

Although this functionality is working now and we've replaced the same heading functionality that we had before, it hasn't really added value yet. The value comes in the next step, when we add data bindings between the parent view and the child-view, creating a flexible heading component that can be used anywhere in the application.

STEP 5 – ADDING THE BINDINGS

In this step, we'll make our `<heading>` custom element flexible enough to handle heading text that is defined in the parent component rather than being hard-coded. To do this we add a `bindable` attribute to the `<template>` element and specify the properties that we want to be able to bind between the parent and child view. In listing 3.13, we set up a two-way data-binding between the parent and child view on the text property. We then render it out to the DOM using the string interpolation binding expression `${text}`. Modify the `/src/resources/elements/heading.html` template as shown in listing 3.13.

Listing 3.13 Heading view modified to include data-binding – heading.html

```
<template bindable="text"> (1)
  <h1 class='page-heading'>${text}</h1> (2)
</template>
```

(1) Set up a two-way data-binding on the text property.
(2) Render the heading text to the DOM

STEP 6 – USING THE NEW ELEMENT

Finally, we need to consume this binding from the parent view. We can do this by using the `propertyName.bind=` binding convention on the heading custom element in the `books.html` view. Modify the `books.html` view to include this data-binding as shown in listing 3.14. In this case, we've bound the literal value `'books'` to the bindable property `text` on the `<heading>` element, but there is nothing to stop us binding to a property on the `books` view-model instead.

Listing 3.14 Modify books view to include the text.bind binding expression – books.html

```
<template>
…
<require from="./heading.html"></require>
<heading text.bind="'books'"></heading> (1)
…
</template>
```

(1) Add the text.bind binding expression to the <heading> element.

This refactoring allows us to encapsulate the UI fragment related to rendering the `heading` into its own element. This element can then be standardized and referenced in any component throughout the application. In this example, this is probably overkill, but it can be very useful in any case where you need to extract only a part of the view without taking along any associated behavior.

3.4 Custom attributes

Custom elements are useful when we want to create an entirely new element in the DOM, but what if we just want to change the way that an existing element behaves? Custom attributes—the key is also in the name—provide a way of implementing new HTML attributes. These attributes can be appended to an HTML element to change the way the element behaves. These HTML elements can be either an in-built element (such as `<input>` or `<div>`) or your own custom element (such as `<book-list>`). These are commonly used to bring third-party libraries into an Aurelia application. For example, we could implement a custom attribute for the Bootstrap tooltip JavaScript library to allow us to easily add a tooltip to any element in our application simply by adding the attribute. For example, if we wanted to add a tooltip to an input element, we'd do `<input tooltip> </input>`. Custom attributes are defined by creating a view-model class that operates on the element to augment its behavior.

The default naming convention for custom attributes is to include a `CustomAttribute` postfix. So, the `tooltip` attribute would typically be called something like `TooltipCustomAttribute`. By using that naming convention, we instruct the Aurelia framework that it should treat this view resource as a custom attribute, meaning that Aurelia should search for any elements in the view that reference that custom attribute name (the name of the class minus the `CustomAttribute` postfix), and construct a new instance of this class, injecting the DOM element that the attribute is attached to. Let's implement a tooltip custom attribute and look at what's required to consume it in the my-books application.

3.4.1 Creating a tooltip custom attribute

The following steps are required to create an attribute:

1. Create a new JavaScript file for the view-model named `attribute-name.js` (for example, `tooltip.js`)
2. Export a class from this file named `AttributeNameCustomAttribute` (for example, `TooltipCustomAttribute`)
3. Implement the behavior that you want (for example, initializing the Bootstrap tooltip).
4. Require the newly created `CustomAttribute` view resource into the view that needs it.
5. Add the new attribute to the target element.

Let's begin by creating a new JavaScript file, `/src/resources/attributes/tooltip.js`, and exporting the view-model class as shown in listing 3.15.

Listing 3.15 First implementation of the bootstrap tooltip custom attribute – tooltip.js

```
import {inject} from 'aurelia-framework'; (1)

@inject(Element)                          (2)
export class TooltipCustomAttribute{      (3)

  constructor(element){                   (4)
      this.element = element;
  }

  attached(){
      $(this.element).tooltip();          (5)
  }

  detached(){
      $(this.element).tooltip('dispose'); (6)
  }
}
```

(1) Import the inject class for dependency injection.
(2) Inject element this attribute is attached to.
(3) Follow the default custom attribute naming convention.
(4) Take the current element reference injected into the constructor.
(5) Initialize the Bootstrap jQuery tooltip plugin on the element.
(6) Dispose the tooltip reference when the view is detached from the DOM.

Custom attribute and custom element view-models can ask which HTML element they're attached to. They do this by asking for a reference to this element in the constructor via dependency injection. This is key, particularly in custom attributes, because we often want to perform an operation on the element that the attribute is attached to. After we've accessed the element in the constructor (which in this case will be the button on which we want to display the tooltip), we can then perform the action needed to augment the elements behavior. In this case, we hook into the attached Aurelia life cycle hook, which is called when the element is attached to the DOM. The reason that we do it here rather than in a different hook (for example the constructor) is that we need to ensure that the element is available before we try to reference it. Within the attached method, we wrap the element in a jQuery object and initialize a tooltip on the element. From there, the Bootstrap JavaScript takes over and performs the steps necessary to attach the tooltip to the element. We also hook into the detached life cycle hook to dispose of this plugin object and clean up the DOM after the element is detached. By creating an ever-increasing number of these plugin references each time we navigate between Aurelia views, this ensures that we don't leak resources.

Now that we've got the new custom attribute defined, it's simply a matter of requiring that element into the view, which we can do using the `<require from='attribute-name'>` syntax. Require the custom attribute into the `/src/resources/elements/book-list.html` view as shown in listing 3.16.

Listing 3.16 Put the tooltip custom attribute to use in the book-list view – book-list.html

```
<template>
  <require from="../attributes/tooltip"></require> (1)
  ..
          <div class="col-11">${book.title}</div>
          <div class="col-1">
          <span class="remove-button"
              click.delegate="removeBook($index)"  (2)
              tooltip
              title='Remove book from list'>       (3)
                  <i class="fa fa-trash" aria-hidden="true"></i>
              </span>
          </div>
  ..
</template>
```

(1) Require the new tooltip attribute into the view
(2) Pass the index of the book in the current iteration to the removeBook method
(3) Add the tooltip and title attributes to the button

We've brought the new attribute into our view and attached it to the `<span>` element. The title for the tooltip is taken from the title attribute on the element (though this is built in bootstrap functionality rather than our attribute behavior). We've also used an Aurelia repeater *contextual property* to pass the current book to remove. Contextual properties are properties made available to the view in certain binding scenarios (in this case a repeater). We'll cover repeater contextual properties in depth in section 3.6.

Next, we'll add a new `removeBook` method to the `BookList` view-model to remove a book from the books array with the index parameter passed from the View `removeBook(index)`. Modify the `./src/resources/book-list.js` file as shown in listing 3.17 to incorporate this method.

Listing 3.17 Modify the BookList view-model to incorporate book removal – book-list.js

```
import {bindable} from 'aurelia-framework';

export class BookList {
  @bindable books;

  removeBook(index){
      this.books.splice(index, 1); (1)
  }
}
```

(1) Remove book from list by index.

If you refresh the page at http://localhost:9000/#/books, you should see the tooltip displayed above the button as shown in figure 3.5.

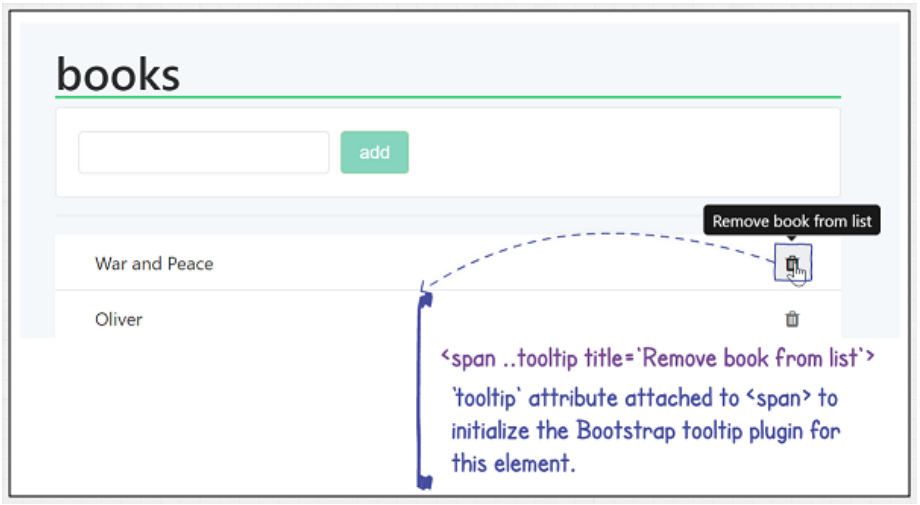

Figure 3.5 book-list.html view updated to use the new tooltip custom attribute that adds a tooltip hover effect on the delete-book buttons.

This tooltip is not limited to this one element, however. We can attach it to any element in the application to easily provide a tooltip.

Currently, there are two major areas for improvement with our new attribute. First, we can display a tooltip only on top of the element; second, we are currently applying the tooltip via a title attribute rather than doing things the Aurelia way and providing this variable as a parameter via data-binding. Let's improve the attribute by resolving both issues.

3.4.2 Single-value binding

In many scenarios building custom attributes, you'll want to bind only a single value. For example, imagine that you have an `uploadable` attribute that transforms any element into a drag/drop upload area. In such a case, you may need to supply only the API path that the file should be uploaded to. Aurelia has a feature for this called *single-value binding*. First, let's modify the `tooltip` attribute in the `tooltip.js` file to take a single value for the title, as shown in listing 3.18.

Listing 3.18 Modify tooltip custom attribute to use a single-value binding – tooltip.js

```
…
export class TooltipCustomAttribute{
 …
  attached(){
      $(this.element).tooltip({title: this.value}); ❶
  }
  …
}
```

❶ Modify the tooltip method to take the title option

The tooltip now takes the `title` as an option rather than this value coming from a `data` attribute on the element. Because this is a single-value binding, we can access the value directly on the custom attribute object using `this.value`.

> **NOTE** This `value` is not set until data-binding is completed on the component, so the value is available in the component life cycle hooks from `bind` onward. We're ok in this case because we are accessing the value in the attached hook, which is fired after binding has been completed.

Modify the `/src/resources/elements/book-list.html` view as shown in listing 3.19 to pass this value along in the custom attribute binding expression.

Listing 3.19 Modify book-list view to use a single-value binding on the tooltip custom attribute – book-list.html

```
<template>
 …
              <span class="remove-button"
                click.delegate="removeBook($index)"
                tooltip='Remove book from list'> ❶
                  <i class="fa fa-trash" aria-hidden="true"></i>
              </span>
 …
</template>
```

❶ Pass the literal title value to the tooltip attribute.

In this case, we're specifying a literal value for the title of the tooltip, but this could also be a binding expression such as `${tooltipTitle}` if we wanted to bind this from the `BookList` view-model.

3.4.3 Options binding

Single-value bindings work well when you've got only one value to bind, but with the `tooltip` we already have two that we know about right now, and there is the potential that we'll want to add more later. To achieve this, we can instead use an *options binding*. Options bindings allow you to supply a set of values to a custom attribute from the view. Let's modify the `tooltip` custom attribute in the `tooltip.js` file as shown in listing 3.20 to allow for configuration of both the `title` and the `position` using an options binding.

With options bindings, you need to create a `@bindable` property on the custom attribute view-model for each value that you want to bind. In this case, we expose both the `title` and `placement` attributes as `bindable` properties, making them available from the view. Again, we're able to use the values for both properties within the context of the `attached` hook, because binding has already completed at this point, passing the values along from the view.

Listing 3.20 Modify tooltip custom attribute to use an options binding – tooltip.js

```
…
export class TooltipCustomAttribute{

  @bindable title;      (1)
  @bindable placement; (2)

  constructor(element){
      this.element = element;
  }

  attached(){
      $(this.element).tooltip({title: this.title,
 placement : this.placement}); (3)
  }
 …
}
```

(1) Add a bindable property title.
(2) Add a bindable property placement.
(3) Initialize the tooltip with option values from the bindable properties.

Following this, we need to modify the view to pass both options. Multiple options can be passed to a custom attribute binding expression using the `customAttributeName='propertyName1: value1; propertyName2: value2'` syntax, where the values are either a literal value or a JavaScript expression. Modify the `/src/resources/elements/book-list.html` file to pass the title and placement along in the binding expression as shown in listing 3.21.

Listing 3.21 Modify book-list view to use an options binding on the tooltip custom attribute – book-list.html

```
<template>
…
              <span class="remove-button"
            click.delegate="removeBook($index)" (1)
            tooltip="title: remove book from list;
                             placement: right">  (2)
                    <i class="fa fa-trash" aria-hidden="true"></i>
              </span>
…
</template>
```

(1) Pass the index of the current iteration into the removeBook method
(2) Pass the literal tooltip options values to the view-model in the binding expression

In this case, we've passed the literal values for both the `title` and the `placement` to the attribute binding expression. If you refresh the browser now and place the pointer over one of the `remove-book` buttons, you should see the same title but the tooltip should now float to the right, as shown in figure 3.6.

Figure 3.6 Remove buttons now use the configurable tooltip to allow a placement and title to be specified as a part of the custom attribute binding expression.

Because we leaned on an existing JavaScript library to do the bulk of the DOM manipulation work, in this case we didn't need to get too advanced with what custom attributes can do. We'll explore custom attributes in more detail later when we create our own autocomplete input custom attribute from scratch, without any third-party JavaScript dependencies. We'll also explore other custom attribute binding options such as *dynamic options bindings,* which will allow us to get a lot more sophisticated with how our custom attribute function.

You should now have a reasonable understanding of two kinds of HTML behavior view resources: custom elements and custom attributes. Next, we'll look at several tools provided by Aurelia's templating engine that allow you to bring your custom elements and attributes to life.

3.5 my-books project status

- To this point, we added a hint of style to my-books using a CSS resource to load custom styles into the app view, combined with the third-party libraries Fontawesome and Bootstrap. Next, we used an HTML-only custom element to encapsulate the header of the my-books page into a component without any non-view behavior. We also decomposed the application into a series of easy to reason about components using custom elements and attributes. The current project state is depicted in figure 3.7.

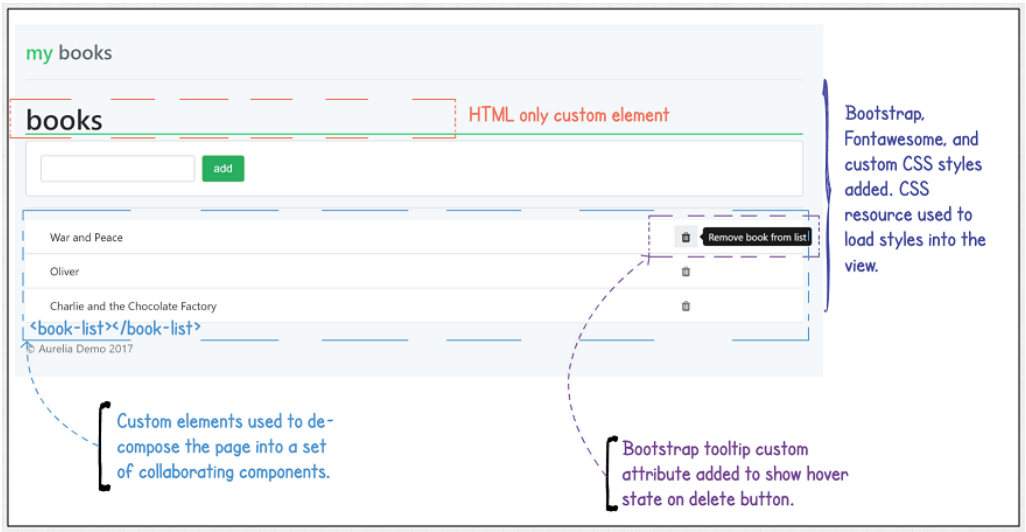

Figure 3.7 Current my-books application book management page status. Styles added, and page de-composed using custom elements and attributes.

You can download the completed chapter 3 my-books project from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

3.6 Summary

In this chapter, we've learned the following:

- Aurelia's templating and binding system is built on a set of core primitives.
- One of these core primitives is the CSS resource, which allows you to load CSS into your Aurelia views.
- HTML behaviors are another kind view resource that allow you to extend the DOM by adding custom HTML elements and attributes.
- Aurelia applications are built by composing custom elements, attributes and other types of components.

Chapter 2 gave you grounding in the basics of what the Aurelia templating and data-binding system had to offer. In chapter 4 we'll build on this foundation, delving into the details of Aurelia's templating and binding system works under the hood. You'll also learn how and when to use the variety of binding tools available in Aurelia's toolbox, from one-way and two-way binding to the binding commands available for handling DOM events.

4

Aurelia templating and data binding

This chapter covers

- Aurelia templating
- Understanding the Aurelia data-binding system
- Handling DOM events

The best user interfaces make it obvious what the user can and can't do at any point in time; they're intuitive. One example of this is the clipboard feature in Microsoft Word. In Microsoft Word, it's not possible to click the *paste* button until you've added something to the clipboard by clicking the *copy* button. This is made obvious by adding a grey hue to the paste button until this option is available.

User interface tweaks like these make the user journey obvious by not requiring users to think when it can be avoided. An alternative approach would be to show an error or warning message when the user tries to paste something without having anything currently on the clipboard, which I'd argue is a much more invasive way of achieving the same result. One of the beautiful things about single-page applications is how easy they make it to add these kinds of UI interactions. In this chapter, you'll learn how to use a combination of templating conditionals, repeaters, and binding commands to build these kinds of interactions into your Aurelia applications.

4.1 Templating conditionals

Template conditionals allow you to conditionally show or hide a fragment of the DOM based on an expression or view-model property. This allows you to design user interfaces that only

show the user what they need at a given point in time. In this section we'll look at Aurelia's template conditionals: `show.bind` and `if.bind`.

The conditional templating options in Aurelia are as follows:

- `show.bind`: Attach this to an element to have Aurelia show or hide the element using CSS.
- `if.bind`: Attach this to an element to have the element added or removed from the DOM based on the result of the binding expression.

We'll make use of both conditional template options as we implement a new component of the application called `book-stats`. This component will eventually be expanded to display various metrics about books we've read and so on, but to start with, let's add a simple counter to display the number of books that we've added after loading the page. What we should see after we've added this new component is a running total of the number of books added, which is shown only after at least one book has been added to the list. Firstly, we need to create a new view and view-model pair for this component. After we've done that, we'll introduce the templating conditionals to control what's shown on the page.

The `BookStats` view-model (listing 4.1) is straightforward for the most part; however, there is one feature that you've not come across yet called *computed properties*. One of the decorators imported into this class is the `@computedFrom` decorator. This decorator allows us to create a property which will be notified for update when either the `originalNumberOfBooks` or the `books.length` values change. This is useful because it prevents Aurelia from needing to have a loop running in the background, constantly checking whether either of these values have been updated to know if this component needs to be re-rendered. As a rule of thumb, you should use the `@computedFrom` decorator whenever you need to do something in the view that depends on multiple properties in the view model, or when you are binding to the result of a calculated expression, such as `bookTitle.length > 0`.

Create a new a JavaScript file `src/resources/elements/book-stats.js` that contains our `BookStats` view-model class, as shown in listing 4.1.

Listing 4.1 book-stats view-model first iteration – book-stats.js

```
import {computedFrom, bindable} from 'aurelia-framework';  (1)

export class BookStats{

  @bindable books;                (2)
  @bindable originalNumberOfBooks; (3)

  @computedFrom('originalNumberOfBooks', 'books.length')   (4)
  get addedBooks(){
      return this.books.length-this.originalNumberOfBooks; (5)
  }
}
```

(1) **Import the @bindable and @computedFrom decorators.**

2. Books is bound from the parent view.
3. Original number of books is bound from the parent view.
4. Add a computed property that depends on both the originalNumberOfBooks and books.length
5. Calculate the number of added books

After you've created the view-model the next step is to create the corresponding `book-stats` view.

Create a new file `src/resources/elements/book-stats.html` and include the template markup as shown in listing 4.2.

Listing 4.2 book-stats view first iteration – book-stats.html

```
<template>
    <div class="card text-center">
        <div class="card-block">
            <p class="card-text">
                <span show.bind="addedBooks"  (1)
                    class="badge badge-primary">
                    new books ${addedBooks}
                </span>
            </p>
        </div>
        <div class="card-footer text-muted">
            Book Stats
        </div>
    </div>
</template>
```

1. Add a show.bind binding to the addedBooks property and render the value.

The `show.bind` expression is used to hide the `addedBooks` badge until books have been added to the list. We're also using the string interpolation syntax again here to render the added books number into the `<span>` content. The way the `show.bind` expression works is that it adds the `aurelia-hide` class to the element that it's been applied to, which has a style of `display:none`, as shown in figure 4.1. This class is removed when the binding expression evaluates to true.

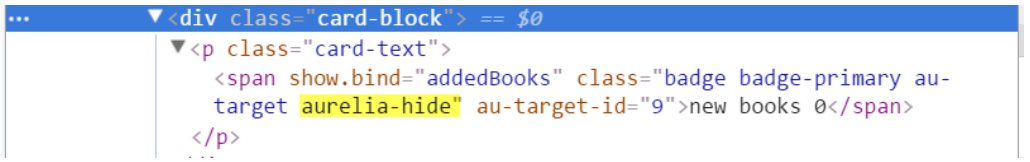

Figure 4.1 Chrome developer tools shows that the aurelia-hide class is added to an element.

A good rule of thumb is to use the `show.bind` conditional unless there is a good reason you need the element to be removed from the DOM entirely. This optimizes templating performance by avoiding unnecessary DOM manipulation.

Now that we've created our `book-stats` component, we need to use it in the books component. We'll do this by following the usual pattern of requiring the new view resource into the parent element and adding a new custom element reference. Modify the `src/resources/elements/books.html` as shown in listing 4.3. You'll notice that we've added another conditional binding to the `book-stats` element—the `if.bind` conditional. This binding ensures that this resource is not initialized and added to the DOM until the books list has been populated.

Listing 4.3 Modify books view to include the book-stats component – books.html

```
<template>
..
<require from="./book-stats"></require> (1)
...
<book-stats if.bind="books.length > 0"  (2)
     books.bind="books"  (3)
 original-number-of-books.one-time="books.length"> (4)
</book-stats>
</template>
```

(1) Require the new view resource into the view.
(2) Reference the element and add an if.bind conditional binding.
(3) Bind the value of the books from the parent to the child view-model.
(4) Bind the original number of books with a one-time binding

A convenient side-effect of using this binding here is that the value of the `originalNumberOfBooks` view-model property is set to 3 once the book list has been loaded, so we have the correct value to send through in the data-binding to the `book-stats` view-model. Following this, we bind the books array between the parent and child-view model (so that it can be used in the calculation), and set up a `one-time` binding on the `originalNumberOfBooks`, setting it to the length of the books array. A `one-time` binding is an example of something called a *binding behavior*. As the name suggests, binding behaviors give you a way of tailoring how a binding behaves to the requirements in each scenario. In this case, we're indicating that the binding should be set only once, which will happen when the component is initialized, and any changes in the value of `books.length` will not be picked up by this binding expression. The `one-time` binding behavior is only one example offered by the Aurelia framework. We'll explore the other options as we look at building more advanced components later.

If you are still following along with the `au run --watch` command keeping the site up to date in the background, you should now see the `book-stats` component rendered as shown in figure 4.2.

Figure 4.2 After including the new book-stats component the my-books book management view should display the number books added in each session.

We've enhanced the my-books application by adding a statistics component using a combination of the `if.bind` and `show.bind` template conditionals. The next enhancement we'll make to my-books application is to add some more interesting styling to the book list, while at the same time learning more about Aurelia repeaters, which we first glimpsed in chapter 2.

4.2 Repeaters

Repeaters are a kind of templating expression that allow you to loop through a view-model collection and render it to the DOM. They're particularly useful for scenarios where you have a list of items (like our book list) or table of data you want to render to your view. Repeaters allow you to iterate over just about any kind of JavaScript collection you can imagine, including the following:

- Arrays `(repeat.for="book of books")`
- Ranges `(repeat.for="i of numberOfBooks")`
- Sets `(let books = new Set(); books.add({title: 'War and Peace'}; repeat.for="book of books")`
- Map `(repeat.for="[bookRating, book] of books")`
- Map `("bookProperty of books | bookProperties")`

NOTE This example relies on a `BookPropertiesValueConverter`. We'll go into value converters later in this chapter.

Let's take another look at this book-list repeater that we created in chapter 2, which is reproduced in listing 4.4.

Listing 4.4 book-list view we created for my-books in chapter 2 – book-list.html

```
<template>
    <ul>
        <li repeat.for="book of books"> ${book.title} </li> (1)
    </ul>
</template>
```

(1) Repeat.for stamps out each of the books.

Repeaters can be used on any element, but in our example, we used it to render each of the books in the list. In this section, we'll expand on the book elements, adding the ability to remove books from the list. In doing so we'll explore some of the other features available to us with repeaters.

Before we begin, let's clean up the `<book-list>` view by removing the `<book-stats>` element. It's not all that useful right now with its naive implementation, but don't worry: it'll see a come-back later when we explore the options of inter-component communication.

Update the `/src/resources/elements/books.html` view as shown in listing 4.5 to remove the `book-stats` element.

Listing 4.5 Remove book-stats element from books view – books.html

```
<template>
 <require from="./book-list"></require>
 <require from="./heading.html"></require>

 <heading text.bind="'books'"></heading>
    …
 <hr/>
 <book-list books.bind="books"></book-list>
(1)
</template>
```

(1) First book-stats component implementation removed for now

Within the repeater, we get access to several variables that give insight into the context of the iteration. Let's look at what options we've got available.

4.2.1 Aurelia repeater contextual properties

Aurelia's binding system makes several properties available to the view within the context of binding scenarios. These are called *contextual properties*. The following properties are available within the context of an Aurelia repeater:

- `$index`: The index within the collection that we're repeating over.
- `$first`: Boolean contextual property that returns true for the first item in the array.
- `$last`: Boolean contextual property that returns true for the last item in the array.
- `$even`: Boolean contextual property that returns true for the items in the array with *even* indexes.
- `$odd`: Boolean contextual property that returns true for the items in the array with *odd* indexes.

I use the `$index` property most often, because it's useful in any case where you need to perform an action on a specific item in the array. In our case, we'll use this to implement the `remove-book` feature, removing an item with the specified index. Let's extend the behavior of the `book-list` element to make use of these contextual properties.

In this example, we're first using the `$even` contextual property to style the even list items differently. This is a somewhat contrived example, because you could achieve the same result through simple CSS. Following this we use the `$first` and `$last` properties to render detail of which is the first and last item in the array. Go ahead and replace the markup in the `src/resources/elements/book-list.html` view, as shown in listing 4.6.

Listing 4.6 Modify book-list repeater to include contextual properties – book-list.html

```
<template>
  <ul class="books list-group list-group-flush">
      <li class="list-group-item ${$even ? 'book-even' : ''}"      (1)
          repeat.for="book of books"> (2)
          <div class="col-11">
               ${book.title} ${bookLocation($first, $last)}</div>  (3)
                        <div class="col-1">
                        <span class="remove-button"
                              click.delegate="removeBook($index)"> (4)
                   <i class="fa fa-trash" aria-hidden="true"></i>
              </span>
          </div>
      </li>
  </ul>
</template>
```

(1) Use the $even context property to give the even rows a different style.
(2) Our standard repeat.for iterator.
(3) Use $first and $last contextual properties to give insight into what element we're on.
(4) The click event of the remove-button is delegated to the removeBook method passing the current $index within the array.

Finally, we pass the `$index` property through to the `removeBook` view-model method to indicate which book should be removed from the array. Although this is a somewhat contrived example (we'd generally not need to use this many contextual properties in one view), it provides you with insight into the kinds of contextual information available within iterators.

It's worth mentioning that even though the binding between the `books` and `book-list` components is one-way, we can propagate the changes back up to the `books` view because we

have a reference to the books array object. When we remove an item by index from the array we are removing it directly from the array of books in the parent component. This communication is achieved by object reference rather than binding. In a real-world application, we'd use a technique such as event aggregator or custom events (covered in chapter 4) to notify the parent component of the change in the child component. This avoids direct object manipulation on the parent, and makes the inter-component interaction more explicit and easier to reason about.

Next, let's finish up this implementation by modifying the `/src/resources/elements/book-list.js` view-model to implement two methods that we've referenced from the view (`bookLocation`, and `removeBook`). Modify this file as shown in listing 4.7.

Listing 4.7 Modify book-list view-model to use contextual repeater properties – book-list.js

```
…
export class BookList {
…

   bookLocation(isFirst, isLast){ (1)

       if(isFirst) return '- first book';
       if(isLast)  return '- last book';

       return '';
   }

   removeBook(index){              (2)
       this.books.splice(index, 1);
   }
}
```

(1) The $first and $last contextual properties that was passed as parameter values from the view.
(2) The $index parameter passed from the view to indicate which item should be removed.

The view should now look like figure 4.3 at http://localhost:9000/#/books.

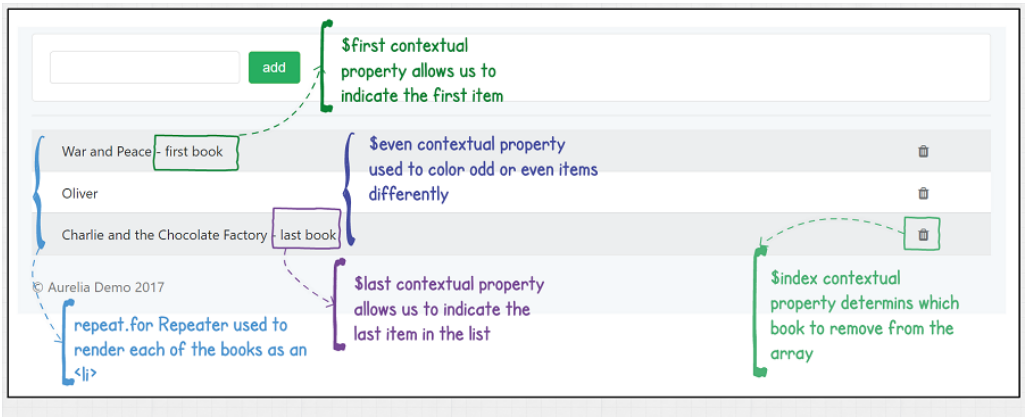

Figure 4.3 my-books book management page re-styled using the contextual properties available within the list repeater such as $first, $last, and $index.

> **NOTE** We've only scratched the surface of the features that repeaters offer. Aurelia repeaters allow you to repeat over other kinds of collections as well, such as ES6 sets or even object properties. The Aurelia Hub documentation has a fantastic page that goes into this in more detail if you're interested in exploring these other options: http://aurelia.io/hub.html#/doc/article/aurelia/templating/latest/templating-basics/9.

4.3 Data binding with Aurelia

Data-binding is Aurelia's way of communicating between the view (HTML) and the view-model (JavaScript). So far, we've used several the basic binding options available in the Aurelia framework, but in this section, we'll take things up a notch. We'll look first at how the Aurelia data-binding system works under the hood. We'll then look at the several types of bindings available, called binding behaviors. By the end of this section you'll know which data-binding to reach for in any given scenario during your Aurelia development workflow.

4.3.1 Understanding how data-binding works

Before we look at the different data-binding features available to us in Aurelia, let's look at how Aurelia implements this functionality. Like other aspects of the Aurelia framework the data-binding system is built on a set of core primitives. Understanding what these primitives are will give you the power you need to use them in many different scenarios and even create your own if you're that way inclined. Figure 4.4 shows a simplified view of how this works in the context of the `<input value.bind="bookTitle">` element. In this section, we'll walk through the different steps of the binding process as detailed in figure 4.4.

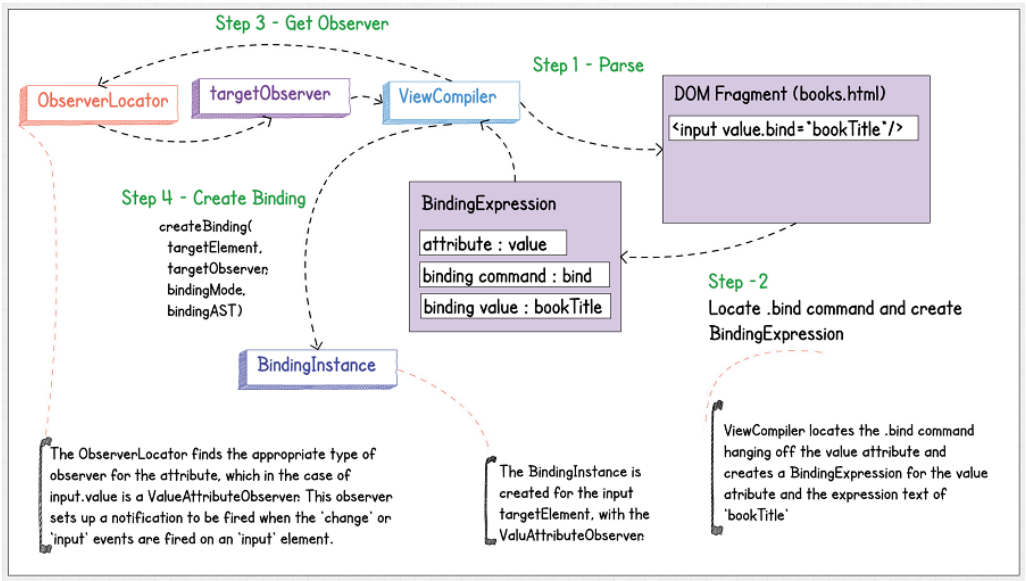

Figure 4.4 The ViewCompiler parses DOM fragments looking for view resources and binding commands. For any binding command located the framework generates a binding expression, which is a factory for generating binding instances.

The Aurelia component responsible for taking a fragment of HTML (our view) and interpreting it to pull out the Aurelia-specific parts is called the `ViewCompiler`. The `ViewCompiler` traverses the DOM tree and searches for `ViewResources`, which we've discussed already (such as `CustomElements` and `CustomAttributes`). It also searches for any fragments that should be turned into `BindingExpressions` by looking for any attributes with a given range of postfix values such as `.bind`, `.one-way`, `.ref`, `.delegate`, `.trigger`, and so on. These postfix values are known as *binding commands*. It also searches for any string interpolation expressions using the `${propertyName}` syntax. Next, the `ViewCompiler` looks at the text value of the attribute or string interpolation, and uses it to build a model (called an Abstract Syntax Tree [AST]) that represents the expression.

> **INFO** **The details of the AST are beyond the scope of this explanation, but if you're interested, you can find a lot more detail in this in-depth article by the current owner of Aurelia's binding engine (Jeremy Danyow) on the Aurelia documentation Hub** http://aurelia.io/hub.html#/doc/article/aurelia/binding/latest/binding-how-it-works/1.

At this point in the binding process the `ViewCompiler` has all the information that it needs to generate a `BindingExpression` from the DOM fragment, including the following components:

- The element that the binding command is attached to

- The attribute on the element (if applicable) that the binding command is attached to
- The value of the binding expression

To understand what this means in practice, let's see what this looks like in the context of the `book-title` input field from the `books` component, shown in listing 4.8. In the context of this element, the `ViewCompiler` knows that it should create a new `BindingExpression` for the `value` attribute on the `<input>` element, and the value should be the result of evaluating the `bookTitle` expression. In this case, the `.bind` `BindingCommand` is associated with the `BindingExpression`.

The `BindingExpression` is in fact a factory for creating binding instances. Before an instance is created, the `ViewCompiler` needs to know what kind of observer it should create for the binding, which it does using the `ObserverLocator`. *Observers* are objects responsible for notifying a view-model object when a value changes in the DOM and vice versa. Different observers are available for the various kinds of HTML elements. The `ObserverLocator` is responsible for picking the correct observer, given the type of element that a binding is applied to.

Listing 4.8 books view fragment demonstrating the data-binding process – books.html

```
<input (1)
    class="form-control"
    value.bind="bookTitle" (2)
    id="book-title"
    type="text">
```

(1) The expression should be created on the input element.
(2) A .bind binding command is found on the .value attribute, with an expression of 'bookTitle'.

For example, in the case of our `<input value.bind="bookTitle">` element, the `ObserverLocator` knows that it needs to create an `ValueAttributeObserver`, which in the case of an input element listens for the `change` and `input` events. A `checkbox` on the other hand uses the `CheckedObserver`, which has its own behavior optimized for responding to state changes on a `checkbox`. This adaptive binding system is one of the things that makes Aurelia's binding engine so fast. Fundamentally, it relies on the concept of observables, where callbacks are set up to fire when a native browser event is received. This avoids the need to constantly check this element value to determine whether it has changed. We'll go over the various kinds of observers in more detail in chapter 5 when we add a more complex `form` to my-books.

Once the `ObserverLocator` has determined the correct kind of `targetObserver`, an instance of the `BindingExpression` is created using the `createBinding` method, passing along the context for the binding, including the `targetElement`, `bindingMode` and so on. The binding instances are then bound into the view.

ADDITIONAL READING If you are interested in reading more about how the Aurelia framework picks up from here and finalizes the process of creating the view instance, I refer you again to Jeremy Danyow's in-depth article on the Aurelia Hub http://aurelia.io/hub.html#/doc/article/aurelia/binding/latest/binding-how-it-works/1.

At this point, binding expressions have the following information:

- The HTML attribute they are attached to (for example `input value.bind`)
- The binding command that they should use (for example `input value.bind`)
- The JavaScript expression for the command (for example `value.bind='bookTitle'`)

These binding expressions are then used to create binding instances when the view is created allowing bindings to be wired up between the view and the view-model.

With this basic understanding of how the Aurelia data-binding system works let's look at the several types of binding commands that the framework offers. We'll do this by tweaking the way that the `<books>` component behaves and adding some simple validation.

4.4 Binding commands

Binding expressions in Aurelia is really a way of connecting HTML or SVG attributes to JavaScript expressions. The syntax for a binding expression is depicted in figure 4.5.

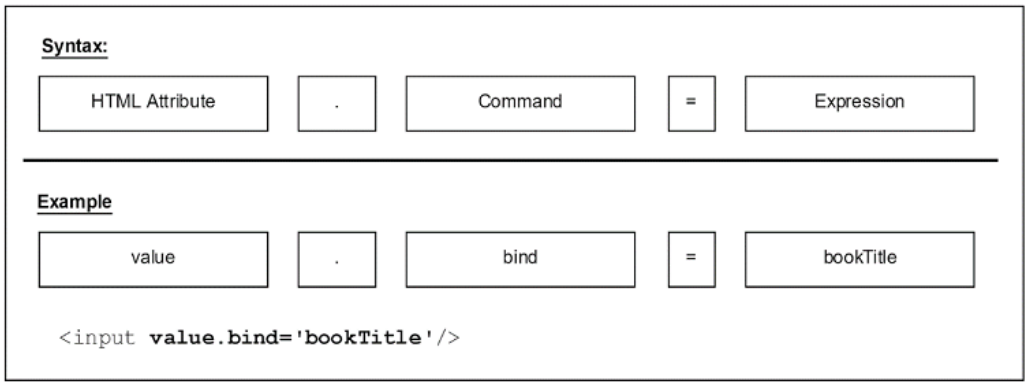

Figure 4.5 The Aurelia binding expression syntax follows the convention of an HTML attribute name followed by a period character, followed by a command name such as *bind,* followed by the equals character, and ending in a JavaScript expression. For example <input value.bind='bookTitle'/>

In figure 4.5, you can see that the binding expression is broken up into three parts:

- HTML attribute: The attribute that we're binding to (in our case value).
- Command: There are several different binding commands available which each serve a different purpose (for example `.one-way` to set up a binding from the view-model to

the view only. We'll go through each of the binding commands available throughout this section. In our case `.bind`.

- Expression: The JavaScript expression that should be evaluated to determine the value. In our case this is a reference to the `bookTitle` property on the view-model.

The type of `BindingCommand` you'll want to use depends on the data-flow requirements between your view and view-model for a given element. Let's go through each of the binding commands in the context of the `<books>` component to see when a given command would be useful.

4.4.1 One-way bindings

One-way bindings should be used when we want data to come only from the view-model to the view and not the other way around. For example, in line with what we mentioned earlier about making the user journey as obvious as possible, let's modify the add button to include a one-way binding on the disabled attribute.

Modify the `books.js` file as shown in listing 4.9 to add a new `canAdd` property on the class. This property will return whether the book can be added based on the length of the `bookTitle` property. In this case, we needed to use the `computedFrom` decorator again to indicate that the binding expression should be re-executed any time the value of the `bookTitle.length` changes. If we left this out, the framework would need to constantly re-evaluate this expression to determine whether the value had changed.

I've structured the code in this way to demonstrate how to use the `computedFrom` decorator. A simpler way to do this is to include the binding expression directly in the view (`input disabled.bind='bookTitle.length === 0'`).

Listing 4.9 Modify books view-model to include one-way binding on canAdd property to constrain whether a book can be added based on the bookTitle length – books.js

```
import {bindable, inject, computedFrom} from 'aurelia-framework'; (1)
import {BookApi} from '../../services/book-api';

export class Books {
  ...
   @computedFrom('bookTitle.length')       (2)
   get canAdd(){ (3)
       return this.bookTitle.length === 0; (4)
   }
}
```

(1) Modify the imports to also include the computedFrom decorator.
(2) Indicate that we want to be notified when the bookTitle.length value changes.
(3) This needs to be a get property because it's calculated.
(4) Calculate whether this is allowed based on title length.

Next, modify the `/src/resources/elemenents/books.html` view as shown in listing 4.10 to take advantage of this new property.

Listing 4.10 Modify books view to include the disabled one-way binding to prevent a book from being added unless a title has been specified – books.html

```
<template>
  ..
  <div class="card">
      <div class="card-block">
  ..
              <input class="form-control" value.bind="bookTitle"
                  id="book-title"
                  type="text">

              <input class="btn btn-success tap-right"
                     type="submit"
                     value="add"
                     disabled.one-way="canAdd"> (1)
        ..
      </div>
  </div>
</template>
```

(1) Set up a one-way binding on the canAdd expression.

We've now added a binding expression that will send data from the view-model to the view (in this case, the disabled status on the button). If you reload the browser at http://localhost:9000, you should see that the button is now disabled until a title value has been specified.

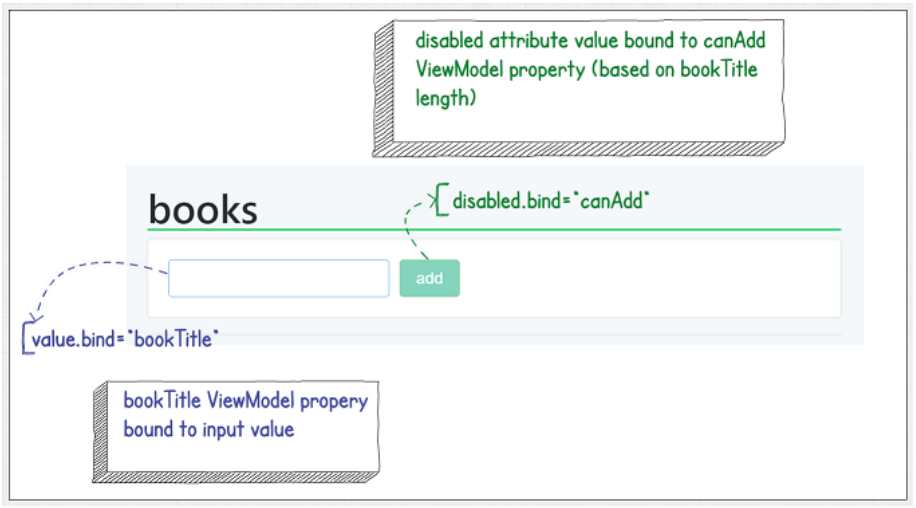

Figure 4.6 The books.html view has been modified to include a one-way binding on the disabled attribute of the add-book button. This prevents books from being added unless a length has been specified.

In fact, we could have gotten away with using just the `disabled.bind` expression due to the reasonable defaults coded into the Aurelia framework. In this case, we don't want to send the `disabled` value back to the view-model; based on its default conventions, Aurelia automatically sets up a one-way binding for us instead.

4.4.2 String interpolation binding

We've already used string interpolation type binding quite extensively throughout the my-books application, so we'll not include a specific example here, but it is worth noting that like the disabled binding we just discussed, the string interpolation binding is automatically one-way only, and sends data from the view-model to the view to be rendered. Here are some scenarios in which string interpolation binding can be very useful:

- Styling elements with the `class` or style `attributes`
- Rendering text from a view-model directly or as a result of pushing it through a value converter.

4.4.3 Two-way bindings

In contrast to one-way bindings, two-way binding expressions send data from the view-model to the view and back. The type of binding that Aurelia infers by default depends on the observer type. For example, the following observer types automatically set up two-way bindings:

- `AttributeValueObserver`: Used to bind to the `value` attribute on elements.
- `CheckedObserver`: Used to bind to the `checked` attribute on checkbox elements.

What this means is that we could have actually written the `<input class="form-control" value.bind="bookTitle"/>` expression like this instead:

`<input class="form-control" value.two-way="bookTitle"/>`,

and it would have the same effect.

To prove that the value of the `bookTitle` is being changed when the input and change events are fired on our `input` element we can add a property observer in the view-model to watch these changes. Modify the `/src/resources/elemenents/books.js` file as shown in listing 4.11 to add a property observer to the `Books` view-model.

Listing 4.11 Modify books view-model to add an observer on the bookTitle property to watch for changes – books.js

```
import {bindable, inject, computedFrom, observable} from 'aurelia-framework'; (1)
import {BookApi} from '../../services/book-api';

export class Books {

  @bindable books;
  @observable bookTitle = ""; (2)
  …
  bookTitleChanged(newValue, oldValue){ (3)
```

```
      console.log(`Book title changed,
                  Old Value : ${oldValue}, New Value: ${newValue}`);
  }
  …
}
```

1. Import the observable decorator to allow us to mark bookTitle observable.
2. Extract bookTitle to be defined outside of the constructor and mark it as @observable.
3. Define a hook to be called whenever the value of bookTitle changes.

We've introduced a new decorator in listing 4.11 called `@observable`. This decorator allows us to hook directly into Aurelia's observation system to observe any variables on our view-model and respond. In this case, we're responding by simply logging the `newValue` and the `oldValue` so that we can see the two-way data-binding live-updating the `bookTitle` property as the `<input>` change event is fired. This can be useful if you need to run a method whenever some variable changes. The syntax for this kind of method hook is `propertyName'Changed'(newValue,oldValue)`. These hooks are available on view-model properties decorated with the `@observable` and `@bindable` decorators.

If you reload the browser and type in the `<input>` box, you should see the values logged to the console in chrome developer tools, similar to figure 4.7:

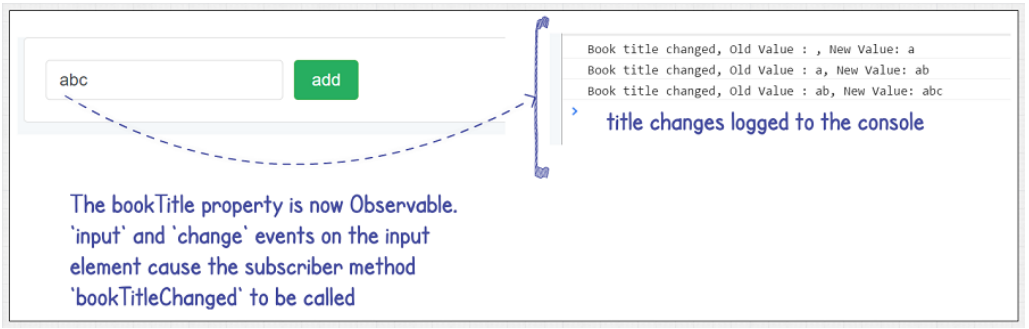

Figure 4.7 Typing values into the input box should now result in these values being logged to the console. Each time the *change* event is fired, the new value of the bookTitle property is logged to the console.

Typing the values a,b,c into the input box results in messages being logged to the console, as shown in figure 4.7.

At this point, we can also check whether the default two-way convention on input elements holds true. If you switch the binding-mode on the input value to `one-way` on in the `/src/resources/elements/books.html` view as shown in listing 4.12, then even when you type a value in the input box, the `add` button should remain disabled, because the view-model has never been notified that the `bookTitle` value should be changed.

Listing 4.12 Modify books View to include a one-way binding rather than two-way – books.html

```
    <input class="form-control"
    value.one-way="bookTitle" (1)
                  id="book-title"
                  type="text"
                  >
```

(1) Switch default two-way data-binding out for two-way temporarily to see the behaviour.

This binding mode output is highlighted in figure 4.8.

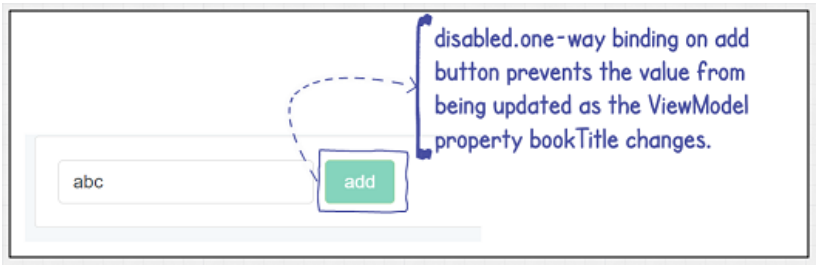

Figure 4.8 With the binding changed to one-way, the disabled value is never updated because we don't receive a changed notification when the value in the input field is changed. Hence, this value is never sent back to the view.

4.4.4 One-time binding command

The one-time binding command is a binding from the view-model to the view that unsubscribes from the property change subscription after the first time that the notification is called. This can be very useful in the following cases:

- Executing the binding has a performance cost (such as retrieving data from an HTTP endpoint).
- You want to calculate the value only once (when the component is initialized and binding is executed). If you recall, we ran into just that case when implementing the `<book-stats>` component. To refresh your memory let's take another look at this binding expression in listing 4.13.

Listing 4.13 books view demonstrating one-time binding – books.html

```
<book-stats if.bind="books.length > 0"
            books.bind="books"
            original-number-of-books.one-time="books.length"> (1)
</book-stats>
```

(1) Set up a one-time binding command on the original-number-of-books property.

In the case of the `originalNumberOfBooks` property on the `BookStats` view-model it makes sense to look at the value the first time the binding is fired (which is when the view-model is instantiated). After that, we don't want to receive any more notifications of changes in the `book.length` property because they would give us an incorrect *original* value.

4.5 Handling DOM Events

Handling DOM events in Aurelia is a piece of cake, and follows the same conventions that you'll now be familiar with based on Aurelia's data-binding conventions. In Aurelia, you can handle DOM events by adding `.delegate` or `.trigger` after specifying an event name on an element (either standard HTML or a custom element), then specifying an Aurelia function to be called when that event is fired. For example, `click.delegate='addBook()'` reads like on click, delegate the clicked event to the `addBook()` view-model method. Figure 4.9 depicts the syntax for handling DOM events in Aurelia.

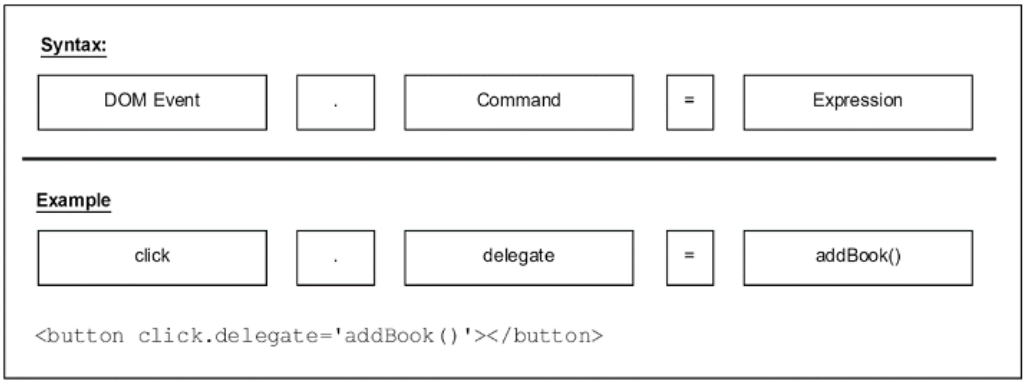

Figure 4.9 The Aurelia event-binding syntax follows the convention of a DOM event name followed by a period character, followed by a command name such as *delegate,* followed by the equals character, and ending in a JavaScript expression. For example <input change.delegate='valueChanged()'/>

Two binding commands can be used to handle events in Aurelia: `delegate` and `trigger`. Let's look at each of these in a bit more detail.

4.5.1 Trigger

The `trigger` command attaches an event handler directly to an element. The expression is invoked whenever the event occurs on that element. Because the `.trigger` command attaches the handler directly to the element that it is applied to, rather than bubbling it up from a nested element, you'll want to use trigger in cases where you need to listen to an event that doesn't bubble (we'll cover why this is the case in section 4.5.2).

Event Bubbling

There are two models for how events are handled in browsers: capturing/trickling and bubbling. These differing models were originated by Netscape and Microsoft who had different opinions on how this should work. It's easiest to understand these differences in the context of an example: If you have an inner element and an outer element and the user clicks the inner element, this click applies to both the inner and the outer element. The question that arose when this functionality was originally being implemented was, which event should fire first? Netscape's opinion was that the event should be fired on the outer element first, and then it would trickle down, firing on the inner element next. Microsoft's opinion was that that the event should be fired on the inner element first, then bubbled up to the outer element. IE versions less than 9 support only the bubbling model, but IE9+ support both the bubbling and capturing/trickling model.

Here are some common examples of the kinds of cases where you would want to use the `.trigger` command:

- `<input focus.trigger = "test()" </input>`
- `<form submit.trigger = "test()"> </input>`
- `<input change.trigger = "test()"> </input>`
- `<input reset.trigger = "test()"> </input>`

You've already come across one example of this type of command during building the my-books application—the book addition form. Let's take another look at this form in the context of the following `.trigger` command:

```
form class="form-inline" submit.trigger="addBook()".
```

In the view, we bound the `addBook` method to the `submit` event on the form. As a result of this, whenever the form is submitted, the `addBook()` method on the `Books` view-model is called. The semantics of the trigger binding command are depicted in figure 4.10.

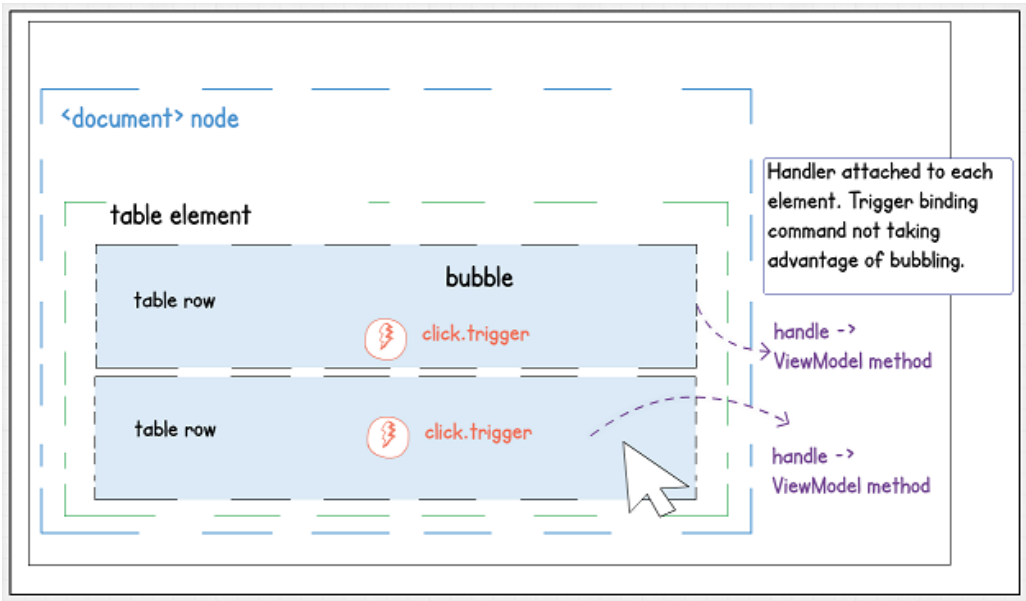

Figure 4.10 trigger binding command is designed to be applied to events that don't bubble, and hence attaches an event handler to each element the command is applied to.

4.5.2 Delegate

The delegate command, in contrast, is attached to a top-level node such as the document node or the nearest shadow DOM boundary. The main benefit of using this is performance. Under the hood, Aurelia can optimize the `delegate` command by aggregating all the delegated event handlers under one handler abstraction, which then forwards the event on to the appropriate target. You can think of this abstraction like a router for your events. It automatically sets up a map between your event target and the handler method, then when the handler is called, it looks up the appropriate method and calls it. This process is depicted in figure 4.11.

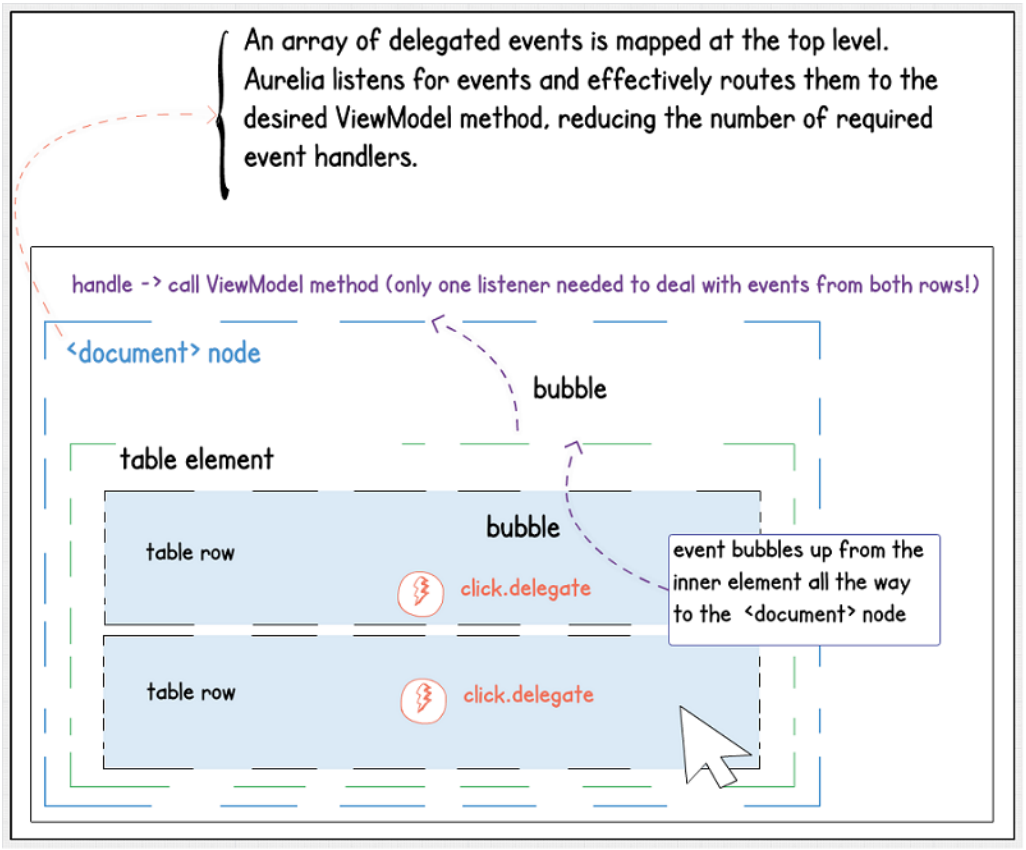

Figure 4.11 click.delegate binding command used to optimize event handling in Aurelia by utilizing event bubbling to minimize the number of required handlers.

A good example of the kind of optimization that can be accomplished is the `book-list` view in which we've already used the delegate binding command to remove books from the list. To refresh your memory, listing 4.14 shows what the relevant part of the `/src/resources/elements/book-list.html` view looks like.

Listing 4.14 Excerpt from the book-list.html view related to the .delegate binding command – book-list.html

```
<li class="list-group-item" repeat.for="book of books">
    <div class="col-10">${book.title}</div>
            <span class="col-1">
        <i class="fa ${book.status | bookStatus}"
           aria-hidden="true"></i>
         </span>
     <div class="col-1">
```

```
        <span class="remove-button"
              click.delegate="removeBook($index)" ❶
              tooltip="title.bind:'Remove book from list';
                       placement.bind:'right'">
              <i class="fa fa-trash" aria-hidden="true"></i>
         </span>
      </div>
  </li>
```

❶ Attach a clicked event handler to the removeBook view-model method using a delegate command.

We've currently got three books in the list, which means if we used the `.trigger` command Aurelia would need to set up three handlers (one for each of the items in the list). This is ok for a small number of items but you can imagine the performance impact if we had hundreds or thousands of items in the list. By contrast, because we are using the `.delegate` command, Aurelia will store a reference to three different event targets in its internal map and call the appropriate handler that corresponds to the target.

> **NOTE** By default, Aurelia always calls `preventDefault()` on any event handled with a `delegate` or `trigger` binding to cancel the event if it is cancellable without any further delegation. If you've used other SPA frameworks in the past that don't do this, you'll be aware of the amount of boilerplate code that this cuts down. In the 1% of cases where you don't want this, however, you can override the default behavior by passing `true` to the event handler function; for example, `addBook(true)`.

A simple rule of thumb for when to use `delegate` versus when to use `trigger` is to use `delegate` unless it's one of those events that don't bubble like the ones mentioned earlier (`change`, `submit`, and so on). There are a couple of exceptions to this rule, so it's best to check the Aurelia documentation on the topic if in doubt.

> **NOTE** The contextual property `$event` is available for these binding commands. You can access it by passing it along in the expression. For example: `addBook($event)`. This can be useful when you want to handle an event that bubbles and apply DOM manipulation such as changing the element style only on the target of the event.

4.6 my-books project status

We've only made one minor adjustment to the my-books application in this chapter, adding a disabled binding to control when users can click the add-book button as shown in figure 4.12.

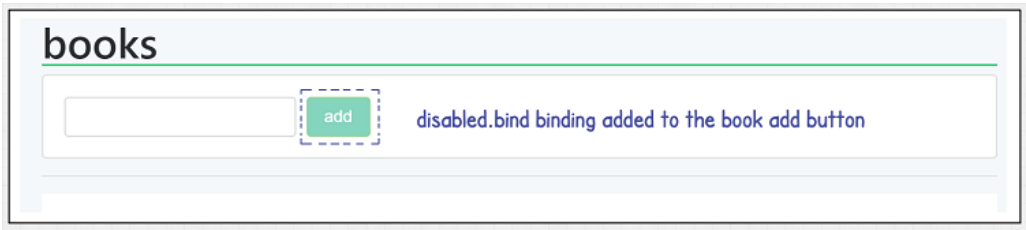

Figure 4.12 Book management page modified to control when users can click the add-book button.

You can download the completed chapter 4 my-books project from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

4.7 Summary

In this chapter, we've learned the following:

- Aurelia applications can be made more intuitive using templating conditionals, which allow you to conditionally show or hide fragments of the view, only revealing what the user needs to see at a given point in their journey.
- Repeaters can be used to iterate over collections such as ranges, arrays or sets and are useful when you need to render a collection from the view-model to the view.
- DOM events such as `input change` or `button click` are handled using the `delegate` and `trigger` binding commands which delegate an event from the view to a view-model method.

Coming up in chapter 5 we'll spice up our my-books views with the addition of value converters and binding behaviors. These tools from Aurelia's binding toolkit put you in control how your view-model data is rendered on the view, what is sent back to the view-model, and how your bindings behave.

5

Value converters and binding behaviors

This chapter covers

- Creating and using simple value converters
- Advanced value converters with parameters
- Value converter composition
- Using Aurelia's built in binding behaviors

So far, the binding expressions that we've looked at have just taken a value from the view-model and rendered it out to the view. But what happens if the value in the view-model isn't suitable for display? For example, if you've got a UTC date in your view-model, you'd most likely want to transform this in some way so that it would be suitable for display in the view. Conversely, you might want to transform a local date entered by the user back to UTC format on the way back into the view-model.

Value converters are an Aurelia view resource responsible for doing just this. Further, what if you need to control when a binding notification is delivered from the view to the view-model? For example, imagine you have an input field bound to a view-model property, and you want only the view-model property to be updated when the user has finished typing in the field? *Binding behaviors* allow you to hook into the Aurelia binding system and change the way that bindings behave to suit your needs in a plethora of scenarios. In this chapter, you'll learn how to use value converters and binding behaviors to gain ultimate control over your Aurelia views.

5.1 Creating a value converter

Let's explore the basic semantics of value converters by implementing a `BookStatusValueConverter` capable of converting from a book status string such as 'good' to a font-awesome icon name such as `'fa-smile-o'`. When we're finished with this value converter, we'll end up with happiness-indicator icons next to each of our books, as shown in figure 5.1.

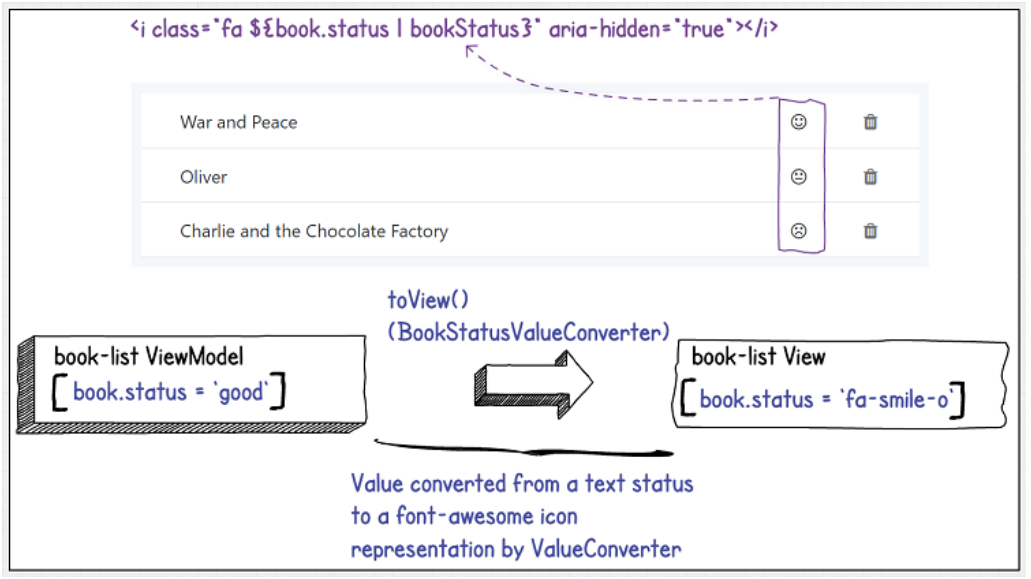

Figure 5.1 The book-list.html view enhanced with a value converter that transforms the book status text from a value like good to a corresponding Fontawesome icon.

Before you create a value converter to take a value from the view-model and transform it for display in the view (and visa-versa), it's a good idea to know the process, which follows:

1. Create a new view-model class named `ValueConverterNameValueConverter` (for example, `DateFormatValueConverter`).
2. Define a `toView(value)` method that takes a value, transforms it, and returns it to the view. This value will likely be a property value on your view-model. Alternatively, create a `fromView(value)` method that takes a value from the view and transforms it before it is updated on the view-model.
3. Require the value converter into the view using the `<require from="../attributes/attributename"` syntax.
4. Use the value converter in a binding expression using the pipe | syntax. For example, `${readDate | dateFormat}`.

Let's start by creating a new JavaScript file, `/src/resources/value-converters/book-status.js`, and add the implementation of the `BookStatusValueConverter` class, as shown in listing 5.1.

Listing 5.1 Create a new value converter that transforms book status text into Fontawesome icons – book-status.js

```
export class BookStatusValueConverter{
  toView(value) {     (1)

      switch(value){ (2)
          case 'bad':
              return 'fa-frown-o';
          case 'good':
              return 'fa-smile-o';
          case 'ok':
              return 'fa-meh-o';
      }

  }
}
```

(1) Implement the toView(value) method to transform the value from the view-model
(2) Switch statement to get the corresponding icon for a given status string

The implementation for this first value converter is basic. We're implementing the `toView` method to take the status value on the book object and return the corresponding icon. The class name `BookStatusValueConverter` is the name of the value converter with a postfix of `ValueConverter` to indicate the kind of view resource this class represents.

Now that we've defined our value converter, the next step is to use it in the view. Modify the `/src/resources/elements/book-list.html` view as shown in listing 5.2 to transform the value of each `book.status` value using the new value converter.

Listing 5.2 Modify book-list view modified to include book-status value converter – book-list.html

```
<template>
  <require from="../attributes/tooltip"></require>
  <require from="../value-converters/book-status"></require> (1)
  <ul class="books list-group list-group-flush">
      <li class="list-group-item" repeat.for="book of books">
          <div class="col-10">${book.title}</div>
          <span class="col-1">
                <i class="fa ${book.status | bookStatus}"
                   aria-hidden="true"></i> (2)
            </span>
            ..
         </div>
      </li>
  </ul>
</template>
```

1. Require the new BookStatus value converter into the view
2. Add font-awesome icon and use the new value converter in the binding expression to render the icon to the view.

If you reload the browser, instead of the raw status-string values, you'll see the relevant font-awesome icon that corresponds to the status of each book. Now that you're familiar with the basics, let's take it one step further by looking at some of the more advanced scenarios that value converters make possible. In addition to simple values, value converters can also be applied to collections. In the next section, we'll add client-side filtering to our book list, and in doing so explore some of the more advanced scenarios enabled by value converters.

5.2 Applying value converters to lists

Value converters can be applied to Aurelia repeaters using the same syntax that we used to apply the value converter to the book's status property `repeat.for='item of items | valueConverter'`. The difference in this case is that the value converter's `toView` method takes the collection the repeater is applied to as its first parameter, `toView(collection, parameter2,..)`.

Imagine that you have an array of numbers `[1,2,3,4]` in your view-model, but you want to render only the even numbers to the view. You could pass your array through an `evens` value converter, which would filter out any of the odd numbers, delivering only even values to `[2,4]` to the view, as depicted in figure 5.2.

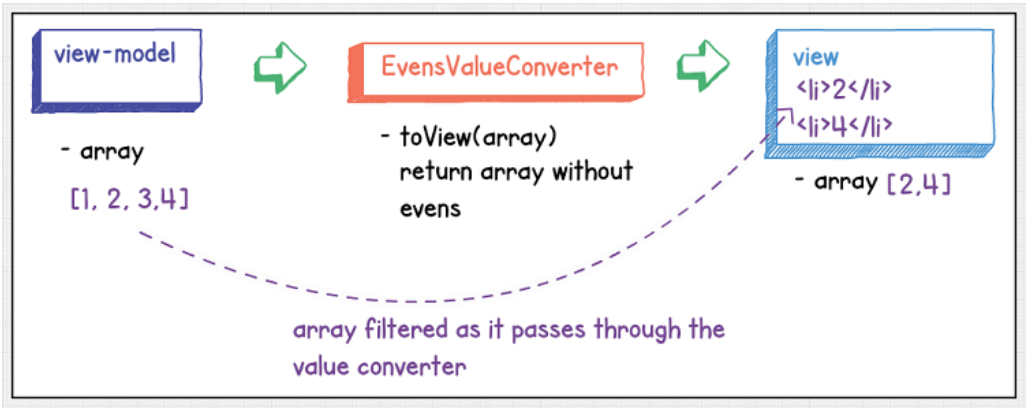

Figure 5.2 You can apply a value converter to an array of numbers to filter only the even numbers to be rendered to the view.

To demonstrate how value converters can be applied to collections in the context of the my-books application, we'll add a new feature to allow users to filter their list of books by typing the book's title into a search term input field. To achieve this, we'll add two new value converters:

- `FilterValueConverter`: This value converter takes the array of books, and the filters out any results where the book's `title` doesn't match the value of the `searchTerm` view-model property.
- `SearchBoldValueConverter`: This value converter takes the book's `title` as input, highlighting any fragments of the title that match the search term.

When we add these new value converters, the updated book-list view will look like figure 5.3.

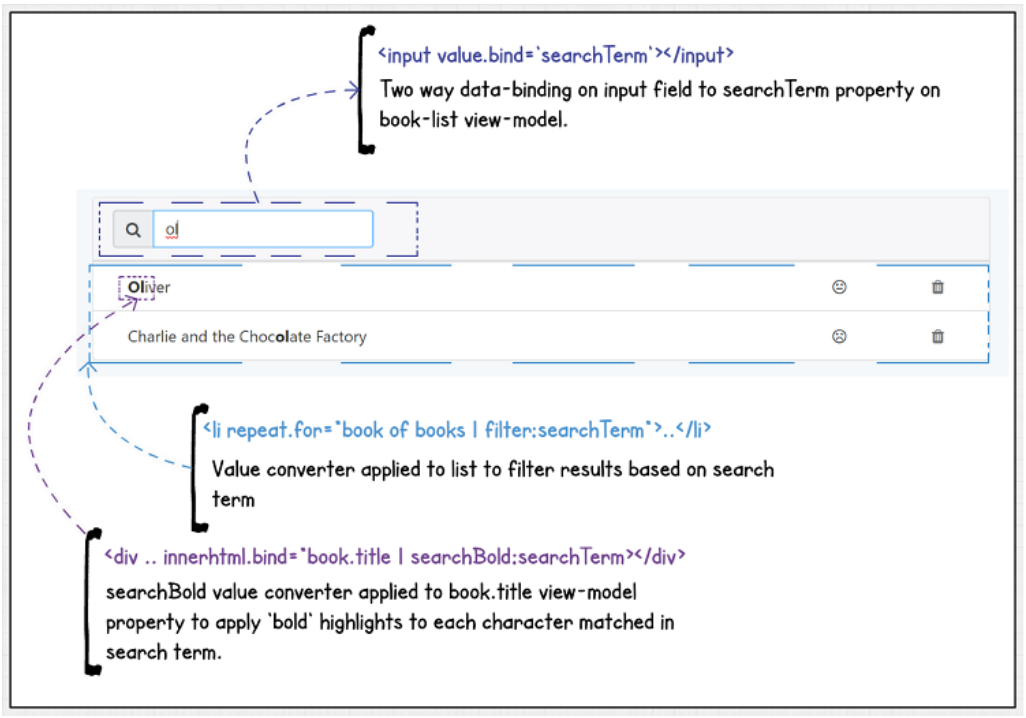

Figure 5.3 Filter and bold highlight value converters have been added to the book-list view to enable users to filter their list of books by typing into the search box.

To begin with, we'll create the `FilterValueConverter`. This value converter's `toView` method takes the `books` array to filter as its first parameter, and a *converter parameter* `searchTerm`, as its second parameter. Converter parameters are values that are passed to a value converter in the view's binding expression. These can be literal values, or values that are bound from the view-model (as is the case with the search term). We'll then use the `array.filter` function to filter out any books where the `title` doesn't match the `searchTerm`. Create the new value converter in `./src/resources/value-converters/filter.js` as shown in listing 5.3.

Listing 5.3 Add the new filter value converter – filter.js

```
export class FilterValueConverter{
  toView(array, searchTerm) { (1)
    return array.filter((item) => { (2)
        return searchTerm
             && searchTerm.length > 0
                ? this.itemMaches(searchTerm,item): true;
    });
  }

  itemMaches(searchTerm, value){
     let itemValue = value.title; (3)

     if(!itemValue) return false;

     return itemValue.toUpperCase()
             .indexOf(searchTerm.toUpperCase()) !== -1;
  }
}
```

(1) toView() function takes the array to filter and the searchTerm to filter by
(2) array.filter() function used filter out any books that don't match
(3) match against the book.title property

With this value converter in place, we'll create the `search-bold` value converter, to add a bold highlight style to any of the book `title` characters that match our search term. As with the above value converter, we also take the `searchTerm` as a converter parameter in this case. Add the value converter `./src/resources/value-converters/search-bold.js` as shown in listing 5.4.

Listing 5.4 Add the new bold highlight value converter – search-bold.js

```
export class SearchBoldValueConverter{
    toView(value, searchTerm) { (1)
       if(!searchTerm) return value;
       return value.replace(new RegExp(searchTerm, 'gi'), `<b>$&</b>`); (2)
    }
}
```

(1) Take the searchTerm converter parameter
(2) Apply the bold highlight

The last step required to implement our new feature is to use the two new value converters in our view. We'll need to add a new search panel to the `book-list` component to allow the user to filter book titles. We'll do this by wrapping the book list in a container div `<div class='card'>`, and adding a new search `input` field with a two-way binding to the `searchTerm` view-model property `value.bind='searchTerm'` . You can pass parameters to value converters using the syntax `.bind='valueExpression | valueConverter:param1:param2…'`. We'll apply the filter value converter to the books array `repeat.for='book of books | filter:searchTerm'`. Next, we'll use the `innerhtml` custom

attribute provided by the Aurelia framework to bind the inner HTML of the book title div to the `book.title` value returned from the `search-bold` value converter `innerhtml.bind="book.title | searchBold:searchTerm"`. Modify the `./src/resources/elements/book-list.html` view as shown in listing 5.5 to use the two new value converters.

Listing 5.5 Use the new value converters in the book-list view – book-list.html

```
<template>
    ...
    <require from="../value-converters/filter"></require>          (1)
    <require from="../value-converters/search-bold"></require>     (1)
    <div class="card">  (2)
        <div class="card-header form-inline">  (3)
            <div class="input-group">  (3)
                <span class="input-group-addon"
                      id="filter-icon">
                    <i class="fa fa-search"
                        aria-hidden="true">
                    </i>
                </span>
                <input type="text" class="form-control"            (3)
                    placeholder="filter"
                    aria-describedby="filter-icon"
                    value.bind="searchTerm">
            </div>
        </div>

        <ul class="books list-group list-group-flush">
            <li class="list-group-item"
                repeat.for="book of books | filter:searchTerm">  (4)
                <div class="col-10"
                    innerhtml.bind="book.title | searchBold:searchTerm">  (5)
                </div>
                ...
            </li>
        </ul>
    </div>
</template>
```

(1) Require new value converters into the view
(2) Wrap the book list in a new div styled "card"
(3) Add a new filter panel with a two-way binding on the searchTerm input field
(4) Run books array through filter value converter passing searchTerm parameter
(5) Bind innerhtml of title to book.title with searchBold value converter applied

You can see the new user search functionality in action by running the au `run --watch` command in your terminal. To build on the example further, what it we wanted to also apply a background highlight the titles of any book that match the search term? We can do this by composing the `search-bold` value converter with another value converter using a technique called *value converter composition*.

5.2.1 Value converter composition

Value converter composition allows you to pipe the output of one value converter into another to transform the source value several ways before rendering it on the view or sending it to the view-model. We can achieve this by simply adding more value converter expressions to a binding expression like so: `<input value.bind='valueExpression | valueConverter1 | valueConverter2' …>` . The input of the preceding value converter is fed into the next value converter. To see this in action, we'll pass the output of the `search-bold` value converter into a new `highlight` value converter in order to apply a background to the title of each matching book. The data-flow for this example will look like figure 5.4. The original book title is first passed through the `search-bold` value converter, and then through the `highlight-value` converter before being rendered into the view.

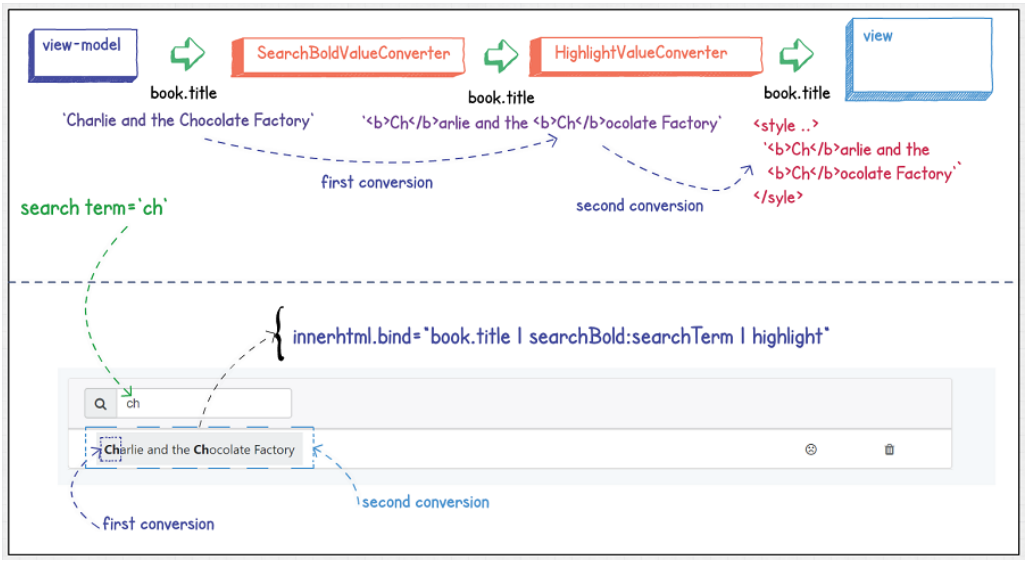

Figure 5.4 search-bold value converter combined with a new highlight value converter to highlight the title of each matching book.

Create the new `./src/resource/highlight.js` value converter as shown in listing 5.6. This converter applies a highlight style to any text containing bold tags.

Listing 5.6 Create the new highlight value converter – highlight.js

```
export class HighlightValueConverter{
    toView(value) { ❶
        if(value && value.indexOf("<b>") !== -1){
            return `<span style=
                    'background-color: #eceeef; padding:10px'>
                    ${value}</span>`;
```

```
        }
        return value;
    }
}
```

❶ Implement toView() method, passing in the text to convert

Next, we'll need to use the new converter in the `book-list` view. Modify the view to require this new value converter, and change the book title binding expression to pipe the output of the `searchBold` value converer into the highlight value converter as shown in listing 5.7.

Listing 5.7 Add value converter composition to book-list view – book-list.html

```
<template>
    …
    <require from="../value-converters/highlight"></require> ❶
     …
        <ul class="books list-group list-group-flush">
            <li class="list-group-item"
                repeat.for="book of books | filter:searchTerm">
                <div class="col-10"
                    innerhtml.bind="book.title | searchBold:searchTerm
                                               | highlight"> ❷
                </div>
                ...
            </li>
        </ul>
    </div>
</template>
```

❶ Require highlight value converter into the view
❷ Compose highlight value converter into binding expression

You can see the new user search functionality in action by running the au `run --watch` command in your terminal.

It's often the little things that can make or break a user's experience with your application. In the next section, we'll look at binding behaviors, a type of view resource that give you fine grained control over how the bindings in your Aurelia views behave. By allowing you to tweak little things like the frequency of binding notifications, binding behavior adjustments can add up to a big difference in application performance, and user experience.

5.3 Binding behaviors

Imagine you want to implement an auto-complete-style search for books that performs a query against the books API as the user enters the name of the book that they want to add to their bookshelf. In this scenario, it would be useful to be able to ignore some of the keystrokes and perform the query against the server only at a certain interval. This wouldn't impact the user experience but would certainly improve the performance of the component by making it less chatty with the back-end API. One way to do this would be to observe the property-

changed notification as we did with the `bookTitle`. In doing this, we'd need to maintain our own state machine to allow the component to remember how long it had been since the last call. We could then execute a request against the server only if a certain amount of time had elapsed since the previous query. This approach has two major drawbacks, however. First, we need to maintain the new state-machine code, and second, we're capturing this event quite late in the pipeline after it has already gone through the entire Aurelia binding system. What if there was a way for this binding to behave so that never actually receives a notification unless that elapsed time has passed?

This is where binding behaviors come in. Binding behaviors are another kind of view resource (like custom element or custom attribute, but most like value converters). These are one of the core concepts, the basic Lego bricks that make up the Aurelia framework. Earlier when we looked at how the Aurelia binding system works, we saw that binding expressions are converted to binding instances to be used in a view. Binding behaviors have complete access to these binding instances throughout their life-cycle. This means that these behaviors can be applied to binding expressions to change the way that they behave. Aurelia comes with several binding behaviors out of the box. In this section, we'll tweak bindings applied in the books component by introducing each of these binding behaviors, and look out how the behavior is altered in each scenario. This will give you an idea of how you can tailor the way your bindings work to suit any problem you may come across when developing Aurelia applications.

5.3.1 Throttle and debounce binding behaviors

The `throttle` and `debounce` binding behaviors give you an easy way to limit how frequently a binding updates. These behaviors are particularly useful when you don't need to know every single time that a value updates. For example, if you were performing an action based on a hover mouse event, or if you were performing a server-side query based on the value in an input box. In these kinds of scenarios, limiting the number of events that fire a notification can give you a nice performance boost.

The `throttle` binding behavior limits either the frequency at which view-model property updates are fixed in `two-way` binding expressions, or the frequency that the view is updated in the case of a `one-way` binding expression.

The syntax for the throttle binding behavior is as follows:

```
value.bind="expression & throttle"
```

By default, this limits the notification frequency to 200ms, but you can change this by specifying the optional parameter as a final argument to the expression, like so:

```
value.bind="expression & throttle:500"
```

This limits the update interval to the millisecond value specified (in this case 500ms). Let's try modifying the `book-title` input-binding expression to throttle the frequency at which we are notified of changes to the `bookTitle` property. Then we can observe the results in the chrome

developer tools. Modify the input binding expression in the `/src/resources/elements/books.html` file as shown in listing 5.8 to apply the `throttle` binding behavior.

Listing 5.8 Modify books view to include a throttle binding behavior – books.html

```
<template>
      …
            <input class="form-control"
                   value.bind="bookTitle & throttle:850" ❶
                   id="book-title"
                   type="text">
      …
</template>
```

❶ Introduce the throttle binding behavior with an interval of 850ms

After making this change, let's reload the view in the browser and try typing in a book title to see how the applied behavior affects what is logged to the console.

As shown in figure 5.5, we now receive a notification and log to the console only when the value has changed to *abc,* because the second character was skipped by the `throttle` binding behavior.

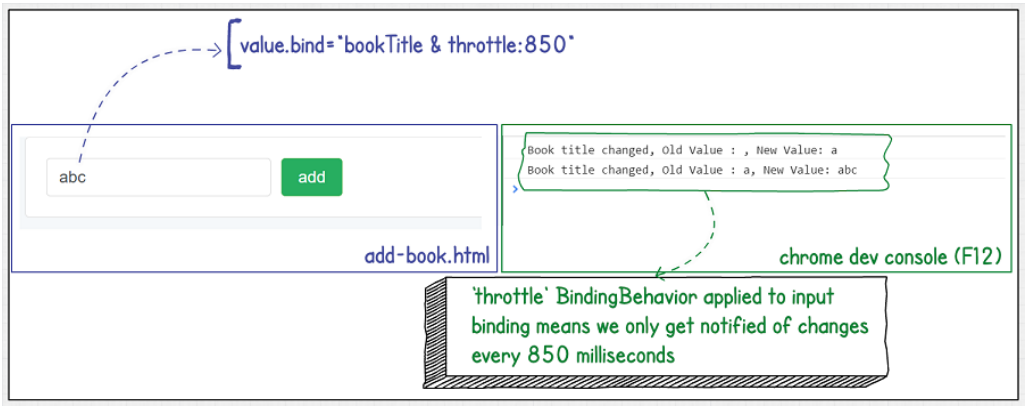

Figure 5.5 Now with the throttle binding behavior applied, only notifications for keystrokes occurring on the given throttle interval are received.

The debounce binding behavior is applied in a way comparable to the `throttle` behavior in that it also limits the notification frequency. In this case, however the notification frequency is calculated based on the amount of time that has passed since the value last changed. As you can imagine, this is very useful for scenarios like an auto-complete style search, where we really want to fire off a request to the server only when the user has paused or stopped typing. The syntax for the `debounce` binding behavior is as follows:

```
value.bind="expression & debounce"
```

This behavior also has an optional final parameter to allow you to control the notification interval. Try updating the input binding expression to use the debounce binding behavior instead. You should see results comparable to figure 5.5. The modified code should look like listing 5.9.

Listing 5.9 Modify books view include a debounce binding behavior– books.html

```
<template>
..
              <input class="form-control"
                  value.bind="bookTitle & debounce:850" (1)
                  id="book-title"
                  type="text">
..
</template>
```

(1) Binding expression modified to include the debounce binding behavior.

Although this sample is somewhat contrived, we'll see a more relevant use of this behavior when we implement the `star-rating` component to allow users to apply ratings to their books in chapter 6.

5.3.2 UpdateTrigger binding behavior

When we looked at how the Aurelia binding system worked end-to-end, I mentioned that the events used to notify a view-model that changes have taken place in a bound element in the view are determined by an Aurelia convention. This convention wires up the event handlers that you're most likely to need by default, so that you don't need to specify them each time. We saw for example that in the case of an `<input>` element, we were notified on the change and input events. But what if we want to be notified on a different event? For example, when applying validation to a form, it's often desirable not to apply the validation message until the blur event is fired, signifying that the user has clicked or tabbed out of that element. Well, it turns out we're in luck. The `updateTrigger` binding behavior allows us to control which events fire a notification. The syntax for this trigger is `value.bind="expression & updateTrigger:blur"`. Let's try modifying the input element to instead use an `updateTrigger` so that the `bookTitle` is updated only when the user navigates away from our input element. Modify the binding expression as shown in listing 5.10.

Listing 5.10 Modify books view to include a updateTrigger binding behavior books.html

```
<template>
..
              <input class="form-control"
   value.bind="bookTitle & updateTrigger:'blur'" (1)
                  id="book-title"
                  type="text">
..
```

```
</template>
```

❶ Binding expression modified to include the updateTrigger binding behavior .

If you re-load this view in the browser, you should see two differences. First, the `disabled` state is not removed from the button until you click or tab out of the `input` field, as shown in figure 5.6.

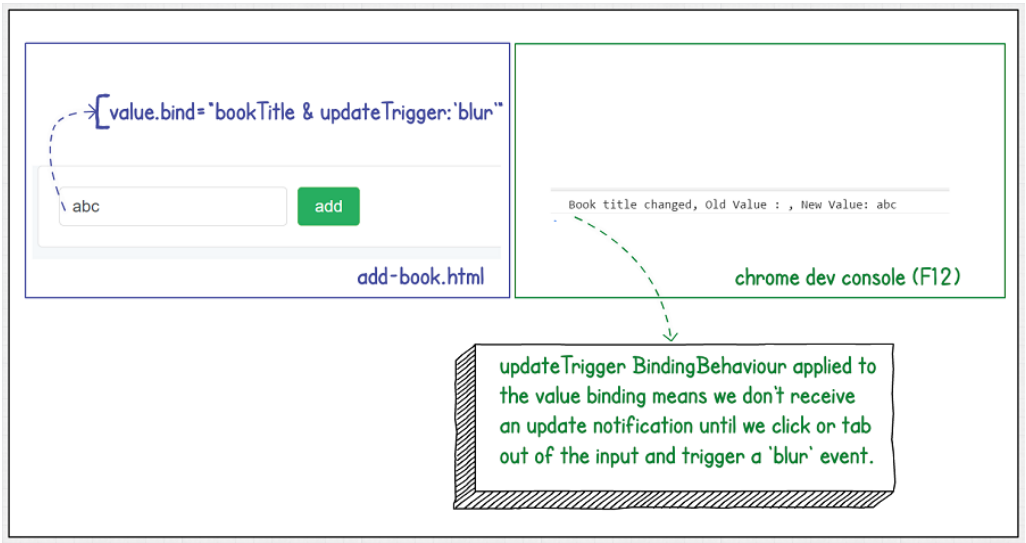

Figure 5.6 The disabled state is not updated until the user tabs or clicks away from the field, because it's now configured to notify of a change only on the blur event. Second, we receive a console log only on blur, meaning we don't see anything until the full abc value is specified.

What if we wanted to also handle the change event in addition to blur? We can do this by adding each of the events that we want to updates trigger on, separated by the colon (:) character - for example, `value.bind="bookTitle & updateTrigger:'blur':'change'`.

5.3.3 Signal binding behavior

The `signal` binding behavior is useful when you need a binding expression to be updated based on a certain event occurring in your application. For example, say we had a field that showed the current time, which we wanted to keep up to date within 1 minute. One way to implement this would be to set a timer in the background to fire an event every minute, and call the signal each time the event was fired. This ensures that we're only re-rendering this fragment of the view when we need to, rather than setting up a constant loop to update the value when we really need to re-render it only once per minute. The syntax for this trigger is `value.bind="expression & signal:'literal-signal-name'"`.

Let's modify the `bookTitle` input element to use a signal `BindingBehavior`. Start by modifying the `/src/resources/elements/books.html` view to include this behavior as shown in listing 5.11.

Listing 5.11 Modify books view to include a signal binding behavior – books.html

```
<template>
…

              <input class="form-control"
                     blur.trigger="refreshSignal()"  (1)
                     value.bind="bookTitle"          (2)
                     id="book-title"
                     type="text">

              <input class="btn btn-success tap-right"
                     type="submit" value="add"
                     disabled.bind="canAdd() & signal: 'can-add-signal'"> (3)
…
</template>
```

(1) Add a blur delegate method to fire our new signal when the user tabs away.
(2) Change the value.bind expression back to the basic two-way binding.
(3) Wire up the canAdd binding expression to signal based on the 'can-add-signal'.

Next, we need to implement the signal in our view-model. Modify the `Books` view-model temporarily as shown in listing 5.12 to implement this signal. In this alternative implementation of the `Books` component, we import the `BindingSignaler`, which allows us to signal the `canAdd` binding to update as a result of calling the `refreshSignal` method. The `bindingSignaler.signal` method takes a name of the signal, which is then referenced in the view (in this case, `'can-add-signal'`). This name does not need to follow any conventions because Aurelia will look for a literal reference to this signal name and connect the signal raised with what is referenced from the view-model. This signal type is most useful in combination with value converters.

Listing 5.12 Modify books view-model to include a signal binding behavior – books.js

```
import {bindable, observable, inject} from 'aurelia-framework'; (1)
import {BindingSignaler} from 'aurelia-templating-resources';   (2)
import {BookApi} from '../../services/book-api';

@inject(BookApi, BindingSignaler)        (3)
export class Books {

  ...
  constructor(bookApi, bindingSignaler){ (4)
      this.books = [];
      this.bookApi = bookApi;
      this.bindingSignaler = bindingSignaler;
  }
  ..
  canAdd(){ (5)
```

```
      return this.bookTitle.length === 0;
  }

  refreshSignal(){
      this.bindingSignaler.signal('can-add-signal'); ❻
  }
  ..
}
```

❶ Remove the @computed decorator import because it's not required in this implementation.
❷ Import the BindingSignaler class.
❸ Inject the BindingSignaler class from the Aurela DI container.
❹ Take an instance of the BindingSignaler singleton.
❺ Remove the @computedFrom decorator and change this into a regular method.
❻ Fire a 'can-add-signal' signal whenever the refreshSignal method is called.

In this section, we've covered a variety of different binding behaviors that Aurelia includes out of the box. With these behaviors under your belt, you should be able to tailor the way that bindings behave to match most of the different binding scenarios that you run into during your Aurelia development life cycle. What happens though if you run into a scenario where none of these behaviors fit perfectly? In that case, it's possible to create your own binding behaviors using custom binding behaviors that work just the way you need them to for a given scenario. We'll cover how to create custom binding behaviors in chapter 12.

5.4 my-books project status

We've added two new features to the my-books book management page. Books can now be filtered using a value converter applied to the books array, with value converter composition used to combine two more bindings to highlight the filtered values. We've also used another value converter to transform the boring string status text on books into a Fontawesome icon for display in the view. The end state of the project in chapter 5 is depicted in figure 5.7.

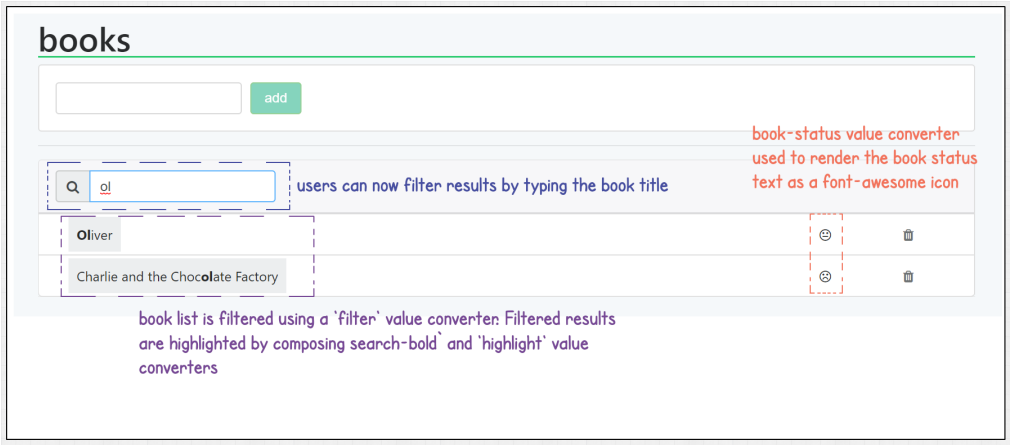

Figure 5.7 my-books book management page state at the end of chapter 5.

You can download the completed project by from GitHub `git clone` https://github.com/freshcutdevelopment/Aurelia-in-Action.git.

5.5 Summary

In this chapter, you've learned the following:

- There are times in your Aurelia development projects where the data in your view model isn't fit for display in the view and visa-versa.
- Value converters give you the flexibility to control how your view-model data is rendered for best usability.
- Value converters can be composed together in a variety of ways by passing the output of one value converter into another.
- You can hook into Aurelia's binding system using built in binding behaviors. This is a terrific way of tweaking settings such as binding notification frequency to improve the performance of your components.

Real-world Aurelia applications are built by composing many well-encapsulated components. In chapter 6, you'll learn how to manage the complexities that come with building an application in this componentized style, using inter-component communication techniques such as the event aggregator and custom events.

A

Installation & set up

This appendix covers

- Installing the Aurelia prerequisites
- Installing the Aurelia CLI

In this appendix, you'll install and set up Node.js and the Aurelia CLI.

A.1 Installing Node.js

If Aurelia is a client side framework, why do we need Node.js? Aurelia applications are prepared for the browser using the Node.js build tools such as Gulp and RequireJS. So, even though Aurelia applications do not require Node.js to run, we do need it to build our project into a deployable package that can be loaded and run in the browser. You can think of this like a compilation step in server-side languages like C, .NET, or Java, where you would need a separate set of build tools on your development machine (such as MSBUILD on the Microsoft .NET side, or Javac on the Java side), as compared with what you would need to run the deployed executable. This transformation process is summarized in figure A.1.

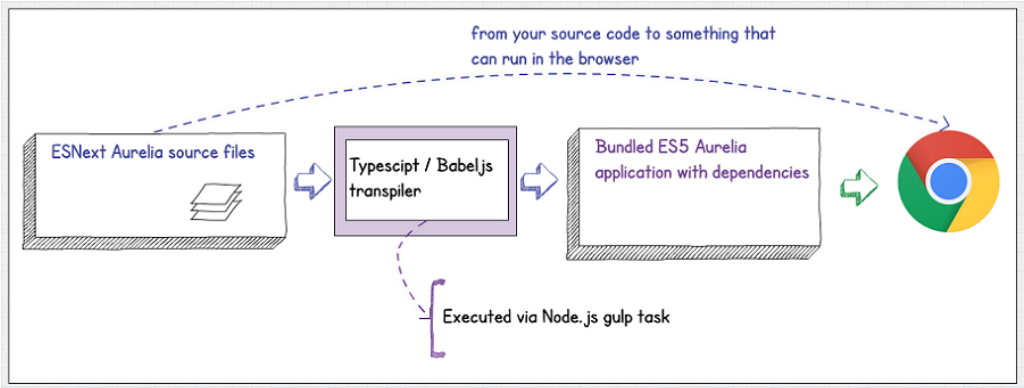

Figure A.1 Node.js is required to take the ESNext source files for your application and transform them into a package that can be run in the browser.

Please download and install Node.js current LTS from the Node.js website https://nodejs.org/en/ using the installation package specific to your environment

OSX

You can download the Node.js installation package for Macintosh from the Node.js website: http://nodejs.org/#download. If you prefer the console you can also install this with Homebrew. https://nodejs.org/en/download/package-manager/#osx

WINDOWS

You can install Node.js on windows by downloading and running the MSI from the Node.js website: http://nodejs.org/#download, or alternatively if you prefer the console this can be achieved using Chocolatey https://nodejs.org/en/download/package-manager/#windows.

LINUX

Node.js installation on Linux is dependent on the distribution. This page on the Node.js website can help you select the most appropriate installation method based on your Linux distribution: https://nodejs.org/en/download/package-manager/#installing-node-js-via-package-manager.

A.2 Installing the Aurelia CLI tool

There are three main pathways for creating a new project with Aurelia:

1. The quick start ZIP download.
2. The Aurelia skeleton projects
3. The Aurelia CLI (selected option)

In this book, we'll use the Aurelia CLI to create and run our project. The Aurelia CLI is a Node.js application that should be installed globally on your development machine. It allows you to do the following tasks:

- Create a new project with the development time dependencies you'd prefer (for example a CSS pre-processor like SASS or plain CSS)
- Download and install NPM packages into your project using NPM or Yarn (a package manager developed by Facebook that improves package installation speed https://yarnpkg.com/en/)
- Build the project, transpiling your ESNext/Typescript source files into ES5 and bundling them for delivery to the browser
- Host the project in a BrowserSync HTTP server that auto-reloads the page when changes are made to your source files
- Unit and integration testing you project

After you have Node.js installed, you can install the Aurelia CLI globally using the following command:

```
npm install aurelia-cli -g
```

A.3 Troubleshooting

If you run into problems with the Aurelia CLI, it's likely due to one of the following causes:

- Unsupported version of Node.js (ensure you're running 6.x).
- Unsupported version of NPM. To use front-end dependencies such as Bootstrap, which are installed via the Aurelia CLI, you'll need a flat directory structure for your installed packages. The flat directory structure was not supported until NPM 3, so it is recommended that you update to this version before creating new projects with the Aurelia CLI. You can check which NPM version you've got installed using the `npm -v` command. If it shows 3.x or greater then you're good to go. If not then you can update to the latest version of NPM using the command `npm install npm -g`.
- Outdated or broken installation of the Aurelia CLI (uninstall and re-install the CLI using NPM – `npm uninstall aurelia-cli -g`, `npm install aurelia-cli -g`).
- Outdated NPM cache: If you receive a message indicating that the `aurelia-cli` package is not a part of the NPM registry when attempting to install the Aurelia CLI, try clearing the NPM cache by running the `npm clear cache -g` command in your terminal. The package should then be picked up and installed correctly the next time you run `npm install aurelia-cli -g`.

A.4 Adding Bootstrap and Fontawesome without a CDN

In chapter 3, we style the my-books sample application with the aid of Bootstrap and Fontawesome. For simplicity, we add these libraries directly via a CDN. However, sometimes

you may want to avoid this method, and load the dependencies using the standard `require` custom element in your views. This section walks through the steps to do this.

INSTALL BOOTSTRAP 4

Let's begin by installing Bootstrap 4 via NPM using the Aurelia CLI. Run the following command to download the node module from NPM and install it into the `node_modules` directory. This command also takes care of modifying the `aurelia.json` file to ensure that the library is included in the vendor bundle:

```
au install bootstrap@4.0.0-alpha.6
```

> **TIP** You may be asked whether you want to install CSS resources when running the au install command, depending on the package being installed. In the case of a library like Bootstrap, you can select 'yes' to have these resources imported and configured in your `aurelia.json` file. Non-visual libraries like Lodash may have CSS libraries purely for unit-testing purposes. In these cases, it's better to select 'no' as you don't want these CSS files bundled with your Aurelia application.

We'll also run the following command to include the Tether library that the bootstrap tooltip JavaScript plugin depends on:

```
au install tether
```

In most cases, just running the commands would be enough, and we'd be ready to start using the library in our project. However, there are some cases where the CLI is unable to automatically complete the set up. One of these cases is when something within the library that you're referencing depends on a global object. The Bootstrap 4 tooltip jQuery plugin relies on the Tether JS library http://tether.io/ to position tooltips relative to an element. It also depends on jQuery. We can still bundle these kinds of legacy dependencies into our `vendor-bundle.js` using a kind of hook built into the bundling pipeline. There are three main steps to the pipeline:

- Prepend: Dependencies included in this step are prepended to the start of the bundle. This is useful for dependencies that rely on an object being available globally (for example Bootstrap).
- Main bundle: The main body of the bundle. Modern libraries or libraries without legacy style dependencies go here (for example `aurelia-fetch-client`).
- Append: Dependencies that rely on a global object defined as a part of the main bundle go here.

Let's start by modifying the `aurelia.json` file to include the Tether dependency in the `prepend` section. Modify the `aurelia.json` file as shown in listing A.1.

Listing A.1 Modified configuration to include Tether in the prepend section – aurelia.json

```
…
"prepend": [ (1)
   "node_modules/bluebird/js/browser/bluebird.core.js",
   "node_modules/aurelia-cli/lib/resources/scripts/configure-bluebird.js",
   "node_modules/tether/dist/js/tether.min.js", (2)
   "node_modules/requirejs/require.js"
  ],
…
```

(1) Prepend section indicates dependencies prepended to the start of a bundle.
(2) Reference the tether JS library from the node_modules folder.

Now that we've got this legacy dependency taken care of, let's reference jQuery and bootstrap to include these in the vendor-bundle as well. Add the dependences shown in listing A.2 to the end of the `dependencies` section.

Listing A.2 Modified configuration to include Bootstrap library – aurelia.json

```
…
},
  "jquery", (1)
   {
       "name": "bootstrap",   (2)
       "path": "../node_modules/bootstrap/dist", (3)
       "main": "js/bootstrap.min", (4)
       "deps": ["jquery"],    (5)
       "exports": "$", (6)
       "resources": [  (7)
          "css/bootstrap.css" (8)
      ]
  }
]
```

(1) Include the jQuery dependency first as Bootstrap depends on it
(2) The name of the library to import.
(3) Dependency source files path (relative to app src)
(4) The main module of the package
(5) Dependencies of the package
(6) The name of the global variable used as the modules exported value
(7) Text resources such as CSS not traceable through the module system
(8) Include bootstrap css resource

With legacy style dependencies such as bootstrap we need to give the bundling system more information as to how it should handle the items (CSS, JavaScript) that need to be included. Key points to note from listing A.2 are that we provided a specific path whether the bundler should look for the bootstrap source code (relative to our application `src` directory), and the where the main module of that package is found (this is what we'll import when using the bootstrap JavaScript plugins). We also indicated the value this main module should be exported as (in this case `$`). This allows us to import the main module from `bootstrap.min`

into a ViewModel using the syntax `import $ from bootstrap`. Finally, the bootstrap library includes CSS which (being text) is not traceable as a part of the module system, as such we need to list it in the resources array so that it will be injected as text into the vendor-bundle.

The next dependency we've got to install is the font-awesome icon library. The steps for this are like the steps we used to install bootstrap. Firstly, use the Aurelia CLI to download the font-awesome package from NPM into our `node_modules` folder:

```
au install font-awesome
```

This automatically wires up a new dependency in our `aurelia.json` file as shown in listing A.3.

Listing A.3 Modified configuration to include font-awesome dependency – aurelia.json

```
{
   …
        "dependencies": [
        …
    v
          {
              "name": "font-awesome",     ❶
              "path": "../node_modules/font-awesome/css", ❷
              "main": "font-awesome.css" ❸
          }
        ]
      }
    ]
  }
}
```

❶ The name of the library to import.
❷ Dependency source files path (relitive to app src)
❸ The main module of the package

So far this should look familiar. We now have the font-awesome dependency installed and bundled as a part of our Aurelia project. We're almost ready to start using these dependencies, but there is one more issue to resolve first. Font Awesome consists of a CSS file and a collection of font files. The problem with what we've set up so far is that the font-awesome CSS looks for the font files relative to the font-awesome CSS file under the directory (for example `../fonts/fontawesome-webfont.eot?v=4.7.0`). In order to work with the Font Awesome directory structure we need to make the font files available under ./fonts, which we can do with the help of a new gulp task. As a refresher from Chapter 1, Gulp is the Node.js based front-end build automation tool used by the Aurelia CLI under the hood to prepare our Aurelia application to be run in the browser. Use the `au generate` command to generate a new gulp task that we'll use to copy the fonts from the node_modules directory to the ./fonts directory as a part of the build:

```
au generate task copy-fonts
```

Modify the newly created gulp task `aurelia_project/generators/tasks/copy-fonts.js` to as shown in listing A.4

Listing A.4 Create new gulp task to copy font files – copy-fonts.js

```
import gulp from 'gulp'; (1)
import project from '../aurelia.json'; (2)

export default function copyFonts() {   (3)
  return gulp.src(project.paths.fontsInput) (4)
    .pipe(gulp.dest(project.paths.fontsOutput)); (5)
}
```

(1) Import gulp build tool,
(2) Import the project node from aurelia.json to access paths.
(3) Create copyFonts function and export it as the module entry point.
(4) Take the fonts files from the source location configured in aurelia.json
(5) Copy the font files to the target location configured in aurelia.json

We've now created a new gulp task that takes the font files from the `node_modules` directory and copies them to the ./fonts directory. The next step is to define the input and output paths that this task references in the `aurelia.json` file. Modify the `aurelia.json` file as shown in listing A.5.

Listing A.5 Modify configuration to include font input and output paths – aurelia.json

```
…
"paths": { (1)
 …
    "fontsInput": "./node_modules/font-awesome/fonts/**/*.*", (2)
    "fontsOutput": "./fonts" (3)
  },
 …
```

(1) Paths used in gulp build tasks.
(2) Directory to copy the fonts from on build.
(3) Directory to copy the fonts to on build.

Now that we've created this task, we need to include it in the build process. We do this by modifying the `build.js` file as shown in listing A.6 to copy the fonts after it has copied the CSS.

Listing A.6 Modify gulp build file modified to include the copy-fonts task – build.js

```
import gulp from 'gulp';
…
import copyFonts from './copy-fonts'; (1)

export default gulp.series( (2)
  readProjectConfiguration, (3)
  gulp.parallel( (4)
    transpile,
```

```
    processMarkup,
    processCSS,
    copyFiles,
    copyFonts ❺
  ),
  writeBundles ❻
);

…
```

❶ Import the new copy-fonts gulp task
❷ Run tasks in series (one after the other)
❸ Firstly read the project configuration (paths and so on)
❹ Run this section in parellel as they are independent
❺ Inlcude the new copyFonts path in the pipeline
❻ Create the bundle files.

Including the `copyFonts` task in the build pipeline ensures this task is run as a part of the build when we run the `au run --watch` Aurelia CLI command.

This completes our setup of the Bootstrap and Font Awesome dependencies. Next, we'll look of how we can make use of the Bootstrap CSS dependency, combined with a custom style to theme my-books.

A.5 Installing the Aurelia validation plugin

You can install the Aurelia validation plugin using the Aurelia CLI using the following command:

```
au install aurelia-validation
```

As with the packages we've installed already, this command downloads the `aurelia-validation` plugin from NPM and configures it in our `aurelia.json` file. Once this has been successfully installed, you'll need to wire up the plugin as a part of the application bootstrap process. As you'll recall, the `main.js` ViewModel file exports a configure function which is called by the Aurelia framework to configure environment settings as a part of application start. This method also allows you to register plugins with the application. Registering a plugin makes the resources within the application (exported classes, custom attributes, binding behaviors and so on, available for import throughout your Aurelia application). With the validation plugin installed, we now need to register it in the `configure` function, as shown in listing A.7.

Listing A.7 Application configure function modified to register aurelia-validation plugin – main.js

```
import environment from './environment';

…
export function configure(aurelia) {
  aurelia.use
```

```
    .standardConfiguration()
    .feature('resources')
    .plugin('aurelia-validation'); ❶

…
}
```

❶ Register the aurelia-validation plugin in the configure function

With the Aurelia-validation plugin installed and registered with our Aurelia application we can now put it to use by adding validation to the edit-books form.

A.6 Setting up my-books server

The my-books SPA has a partner application, the my-books REST API. This replaces the fake backend and `books.json` seed file from chapter 6 onwards. This application consists of a REST API built on Node.js and Express.js, with a MongoDB database. This section walks through the steps required get the my-books REST API up and running on your machine. The setup includes the following steps:

1. Download and install MongoDB
2. Clone the my-books server repository from Github
3. Run the NPM command to start the MongoDB server
4. Execute the NPM command to initialize the my-books server and seed the database

A.6.1 Installing MongoDB

INSTALLING MONGODB ON OSX

If you are on a Mac, please follow the quick start guide on the MongoDB website https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/.

INSTALLING MONGODB ON LINUX

You can install MongoDB on linux by following this quick start guide on the MongoDB website https://docs.mongodb.com/manual/administration/install-on-linux/.

INSTALLING MONGODB ON WINDOWS

1. Download the installer from the MongoDB website https://www.mongodb.com/download-center?jmp=docs&_ga=1.72433297.345738496.1491724167
2. Run the MSI

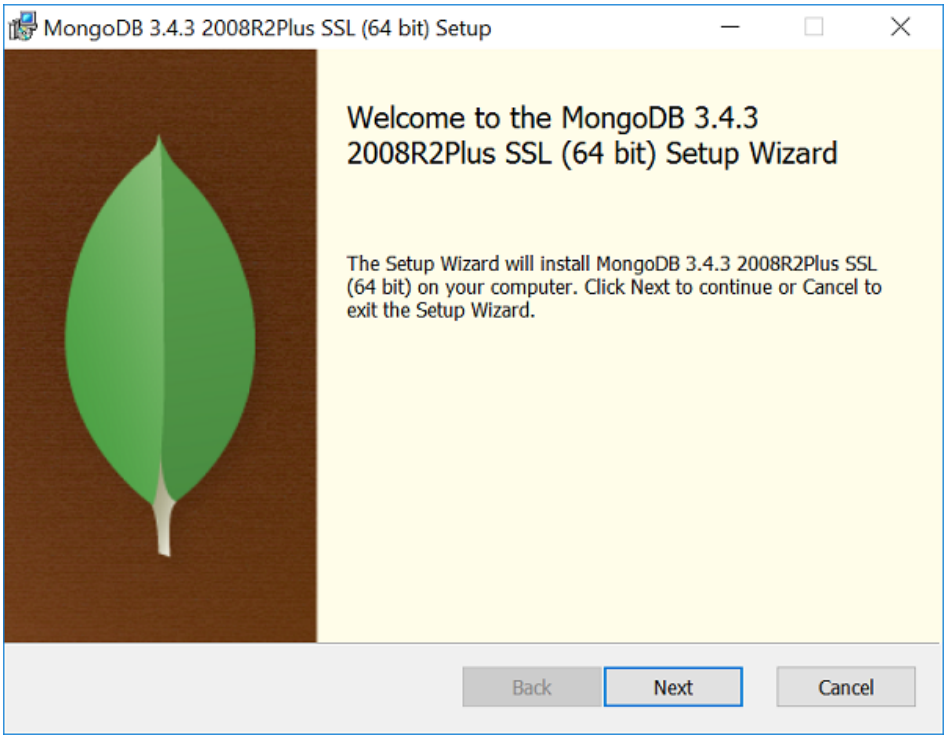

Figure A.2 Run the MongoDB MSI on your machine

3. Add the path to the installed MongoDB binary to your system path
4. With MongoDB installed on your machine. This allows you to run MongoDB by running the `mongo` command on the command prompt.

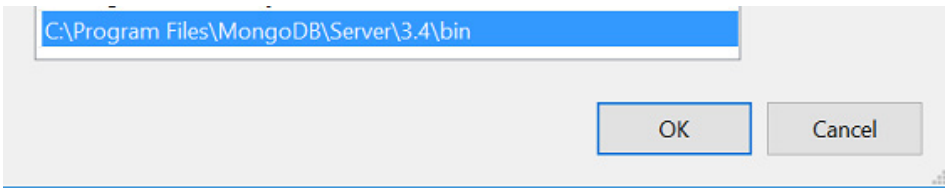

Figure A.3 Modify your step path to include MongoDB binary

A.6.2 Clone the my-books server repository

Clone the my-books-server repository from Github using the following command:

```
git clone https://github.com/freshcutdevelopment/my-books-server.git
```

A.6.3 Start MongoDB server

I've created an NPM command to allow you to easily launch the MongoDB server, creating the my-books MonboDB instance if it doesn't exist at the root of your my-books server application. Before you run this command please launch your terminal and navigate to the directory you cloned the repository into. Then run the following NPM script to launch MongoDB:

```
npm run database
```

A.6.4 Start MongoDB server

The my-books-server REST API runs off a simple Express.js website. You can run this website by running the following NPM script, launching the server at http://localhost:8333:

```
npm run dev
```

This script also seeds the database with the initial my-books seed data on first run. Once the site is running you can running you can test the various API endpoints by installing Postman (available from this website https://www.getpostman.com/), and loading the Postman collection available as a part of the my-books-server Github repository https://github.com/freshcutdevelopment/my-books-server/blob/master/mybooks.postman_collection.json.

A.6.5 Install aurelia-http-client package

The Aurelia framework comes with two HTTP clients: The `aurelia-fetch-client` package and the `aurelia-http-client` package. The `aurelia-fetch-client` comes pre-installed with the default project creation wizard in the Aurelia CLI, but the `aurelia-http-client` needs to be installed separately as it is seldom required in Aurelia projects. You can install the aurelia-http-client using the following Aurelia CLI command in your terminal:

```
au install aurelia-http-client
```

After installing the package, kill your currently running Aurelia application if it's running, and re-run the `au run --watch` command to ensure that the new dependency is included in your vendor bundle.

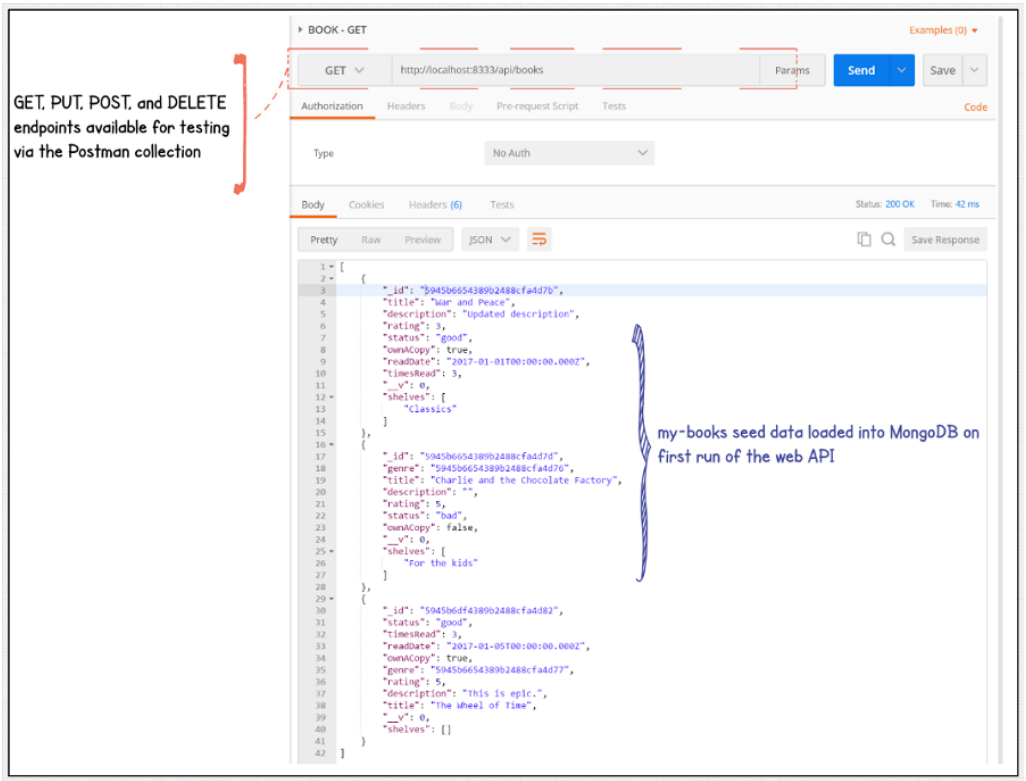

Figure A.4 Sample Postman collection can be used to test the my-books-server endpoints such as book GET, POST, PUT, and DELETE.